Report No. UIUCDCS-R-72-529

SOUPAC SYSTEM PROGRAMMER'S GUIDE

by

Paul Chouinard

with Forward by

Kern W. Dickman

June 28, 1972

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

Report No. UIUCDCS-R-72-529

SOUPAC SYSTEM PROGRAMMER'S GUIDE

by
Paul Chouinard
with Forward by
Kern W. Dickman

June 28, 1972

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

DEDICATION

This report is dedicated to the memory of my father, Carroll B. Chouinard, 1907 - 1972. An editor most of his life, he understood the significance of the written word. I apologize to him for the shortcomings found herein.

FORWARD

SOUPAC programs classify quite naturally into two categories: (1) procedures programs; and (2) system programs. The procedures programs are familiar to research persons from inspection of the SOUPAC reference manual (Report No. UIUCDCS-R-72-370-4). The system programs, on the other hand, are not conspicuous to the users. Indeed there are many users who are scarcely cognizant of their existence. Nevertheless it is the systems programs which control the boundless flexibility of SOUPAC and, at the same time, free the users from needless restraints. In short it is the system programs which give SOUPAC its unique characteristics.

The systems programs were not developed hastily. In fact the basic pattern was set down in 1962 for an IBM 7090. During this period many persons contributed to the evolutionary development of the system and the system routines.

In 1969, however, the development took a rather saltatory character due to the impending conversion to an IBM 360 computer. From then until the present Mr. Paul Chouinard has been the prime mover in the design and programming of the SOUPAC system routines. It is also Mr. Chouinard who has written this document which is a description for the most part of his own efforts. Members of the SOUPAC group and the users, past, present, and future, are indebted to Mr. Chouinard for his diligent and originative achievement.

Kern Dickman
SOUPAC project
June, 1972

ACKNOWLEDGMENTS

To mention everyone who has helped in the development of SOUPAC would be an impossible task. I would still like to thank them all, however, especially the SOUPAC users and the SOUPAC staff. In addition, I would like to acknowledge the following persons who were of particular help to me: Prof. Kern Dickman, Larry Chace, Laurie Parker, Jim Omundson, Bill Walter, and Joe Kolman.

TABLE OF CONTENTS

I. Introduction

A. SOUPAC system summary

SOUPAC is a library of statistical, numerical, and general data handling programs which have been organized into a single system. The minimum hardware configuration to run the SOUPAC system is an IBM 360 with the complete instruction set and 256K bytes of main memory. SOUPAC requires direct access storage for the program library (approximately 600 tracks on a 2314 for the complete library), direct access storage for the SOUPAC system monitor (4 tracks on a 2314), and direct access capability for run-time scratch data sets (size varying with the workload). Recommended, but not essential, is additional direct access space (approximately 50 tracks on a 2314) for macro and subroutine libraries. These two libraries are not referenced during the execution of a SOUPAC job step, but their existence greatly simplifies construction of the program library and the SOUPAC system monitor. If the activity file SOUPLOG is maintained, an additional 5 tracks on a 2314 are required.

The SOUPAC system was designed under a software environment which included OS/MVT, the IBM FORTRAN G & H compilers; and the IBM FORTRAN subroutine library with the extended error monitor facility. The main bulk of code in SOUPAC is written in FORTRAN IV; remaining code, typically SOUPAC system control and I/O support routines, are written in 360 assembly language. All input and output involving user data is done using the standard IBM FORTRAN library I/O routines. Therefore, all input and output of user data follow standard IBM FORTRAN conventions with the following two exceptions:

1. A data field which is entirely "blank" and which is being input under E or F format will be input into a single precision variable as 80000000 hexadecimal (8000000000000000 hexadecimal if double precision). A single precision variable which contains 80000000

(or a double precision variable which contains 8000000000000000) hexadecimal will be output as -0. under E or F format control.

2. User specified formats have an upper bound on the number of characters in the format. All formats must contain 592 or fewer characters; certain formats, where noted, have an upper bound on the number of characters which is less than 592.

For a description of the various procedures available within the SOUPAC program library, please refer to the current edition of DCS report No. 370.

B. Anatomy of a SOUPAC job step

The running SOUPAC system consists of two parts, the executive monitor and the program library. The macro and subroutine libraries are never referenced during execution of a SOUPAC job step and exist only for the purpose of helping build the executive monitor and program library.

A SOUPAC job step is begun by invoking the monitor (see section II.A). This is typically accomplished by using a cataloged procedure which has been set up for the purpose. The monitor is permanently resident throughout the remainder of the job step. Control of loading from the SOUPAC program library, and control of I/O during the job step is through the executive monitor.

The remainder of the running SOUPAC system is brought into main memory, one program at a time from the program library as required, via a LINK macro instruction. The first program invoked by the monitor is always the Syntax Interpreter. The Syntax Interpreter reads the SOUPAC user's program (i.e. parameter deck written according to the rules of the SOUPAC language). From reading this deck, the Syntax Interpreter creates a loading queue which the monitor uses as a list of programs to be sequentially invoked. This execution queue defines the order of the work to be done for the rest of the job step. If the same program is requested several times in a row (e.g. CORRELATIONS), a new copy of the program is invoked from the program library each time. The program library exists as a partitioned data set with each program in the library identified by a unique eight character member name. Each program in the library is a linkedited executable load module, ready to go.

Figure 1 represents what a SOUPAC job step "looks like" in execution and how control flows between the several logical blocks. This figure

is not intended to be complete to the smallest detail, but it is nevertheless fairly comprehensive and serves well as a basic reference on the structure of SOUPAC.

The figure is separated into three parts by horizontal dashed lines. The top section is OS/360, for which SOUPAC assumes no responsibility. The middle section is the executive monitor called UISOUPAC which is brought in at the beginning of the job step and remains resident to the end of the job step. The bottom section represents an arbitrary program which has been invoked from the SOUPAC program library by the loader portion of the monitor.

The figure is also organized about an imaginary line running top to bottom through the center of the page. The left half of the figure represents the flow of program control; the right half of the figure represents the flow of input and output control. Since double arrowheads in the figure represent calls, and single arrowheads represent returns, the left half of the figure (i.e. program control) calls away from OS and returns in the direction of OS, while the right half of the figure (i.e. input and output control) calls in the direction of OS but returns away from OS back to the running program.

Most blocks in the diagram represent more than just the piece of code after which they are named. For example, the block labeled ROWIN can be considered to include subroutines READ, FREAD1, CHKERR and SEQCHK. The block labeled MAIN can be considered to include ALLOC8, whenever ALLOC8 is part of the module. The block labeled NEXT is assumed to include the subroutine TYPE. The block labeled IBCOM# can be considered to contain the format conversion routines IHCFCVTH (also referenced as ADCON#) and BCNVT, the traceback routine IHCETRCH, the interrupt handler IHCEFNTH (also referenced as ARITH#), and interrupt handler support code IHCVOPT

and IHCFOPT (also referenced as ERRSET, ERRSAV, and ERRSTR). The block labeled I/O monitor can be considered to include DADSET, OUTSET, FIOCS#, DIOCS#, IHCERRM, and IHCUATBL. All subroutine names beginning with IHC are control section names of IBM written software; all subroutine names ending with # are entry points to IBM written software. Code such as IBCOM#, FIOCS#, DIOCS#, and ADCON# are discussed within this report by their #-names rather than their IHC-names since the #-names are the commonly used forms used within assembly language code.

Some paths are not represented in the figure. For example, a connection could be made between NEXT and INTRFACE showing WRITE request to FIOCS# and calls to the LOGIN routine in the I/O monitor.

The reason that the monitor and the program library are separated into two different data sets is to facilitate maintenance of the program library and simultaneously to make the system easy to invoke. By separating the two parts it is possible to place the monitor into the 360 system's LINKLIB, and separately maintain the ever-changing program library. If the entire SOUPAC system were in LINKLIB, it would represent a burden to the entire 360 system maintenance. On the other hand, if the monitor were not in LINKLIB, each SOUPAC user would need to have a JOBLIB dd-card at the beginning of his job. By having its own unique program library, the SOUPAC staff is able directly to maintain, autonomously, its own system. Since the monitor remains relatively stable, its presence in LINKLIB does not present a burden to other 360 system operations.

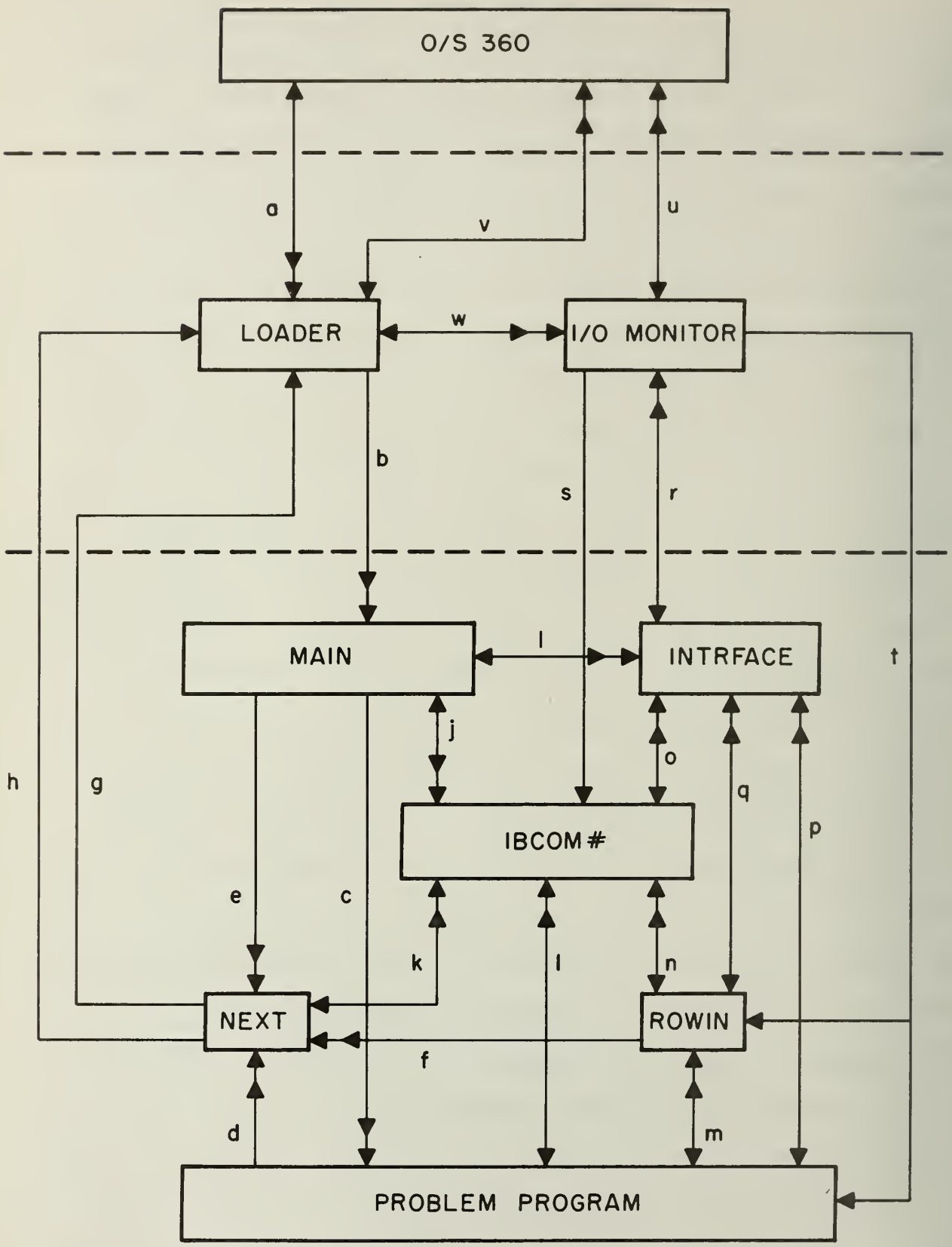
O/S 360
a
v
u
LOADER
w
I/O MONITOR
b
s
r
MAIN
l
INTRFACE
t
j
o
h
g
q
p
IBCOM#
e
c
k
l
n
NEXT
ROWIN
f
d
m
PROBLEM PROGRAM

Figure 1

Key to FIGURE 1.

Program control

a. UISOUPAC invoked by OS to begin the job step; UISOUPAC returns to OS to terminate the job step.

b. Loader calls program from program library via LINK macro instruction.

c. MAIN calls the problem program.

d. CALL NEXT or CALL ERROR from the problem program.

e. CALL NEXT, CALL ERROR, or CALL FORTERR from MAIN or ALLOC8.

f. CALL ERROR from ROWIN, READ, or FREAD1.

g. Normal return to loader thru supervisor corresponding to call at b. above.

h. Direct return to loader (not thru supervisor) in case of error termination or if current program is the last one executed for the job step.

I/O control

i. Call and return to write out the number in the execution queue of the current program and to write out the name of the current program. Also, call and return to write out error message from ALLOC8, if necessary.

j. Call and return to initialize IBCOM# and to read into IPAR from DSRN 3.

k. Call and return from TYME to print out CPU time for the current program.

l. Call and return to IBCOM# resulting from I/O statements in problem programs.

m. Calls to and returns from ROWIN, ROWOUT, TWOIN, TWOOUT, and HEADER.

n. Calls to and returns from IBCOM# for READ, WRITE, and REWIND statements in ROWIN, READ, and FREAD1.

o. Calls to FIOCS# and DIOCS# from IBCOM#; return for READ from DSRN 5 and WRITE to DSRN 6.

p. Calls to FIOCS# and DIOCS# from problem programs; return for WRITE to DSRN 6. Also, calls for DABTBL, DADSET, and other I/O monitor functions.

q. Calls to OUTSET and COPY; return for call from COPY.

r. Transmission of calls to the I/O monitor; return is for READ from DSRN 5 or WRITE to DSRN 6.

s. Return for calls originating in IBCOM#, o. above, which were not a READ from DSRN 5 or a WRITE to DSRN 6.

t. Return for calls at p. and q. above not otherwise returned to (e.g OUTSET and DADSET).

u. SVC's for BSAM and BDAM.

v. SVC's for QSAM in loader. Also, note implicit BPAM in b. above.

w. Calls to and returns from I/O monitor to close I/O monitor controlled data sets.

Note: double arrowheads represent calls; single arrowheads represent returns.

C. Constructing the SOUPAC system

Assuming a complete set of "correct" source decks, the following procedure will result in a SOUPAC system.

1. Using the IBM utility IEBUPDAT, construct the macro library. This step must be performed first. At the University of Illinois this data set is named USER.P0098.SOUPMAC.

2. Create the subroutine library. Assemble and compile the various subroutines. Assemblies will require the concatenation of the 360 system macro library and the SOUPAC macro library. Linkedit the amassed assemblies and compilations into separate partitions of a partitioned data set. When link-editing, specify PARM='LIST,MAP,NCAL' for the linkedit step. At the University of Illinois this data set is named USER.P0098.SUBLIB.

   Current 360 restrictions at the University of Illinois (size of the SYS1.SYSJOBQE data set) renders a straightforward construction of the subroutine library impossible within a single job. For simplicity of maintenance of the subroutine library it is not desirable to break construction of the library into separate jobs. An alternative method for constructing the library by using two consecutive linkedit steps in a single job was originally suggested by Rick Wells, formerly of the Department of Computer Science 360 programming staff. Mr. Wells' suggestion consists of constructing the subroutine library by a single assembly, a FORTRAN G compilation, a FORTRAN H compilation, a linkedit step to format the object modules into a linkedited structure, and a final linkedit step to put each subroutine into its separate partition. Both linkedit steps require PARM='LIST,MAP,NCAL' to be specified, and the second linkedit step requires input of linkedit commands which have been created by the assembly of a specially written macro LKED (see section VI.B). NAME and ALIAS statements are provided by the LKED macro for correctly assigning references for each partition.

3. Create the monitor. Creation of the monitor requires an assembly, a FORTRAN compilation and a linkedit. The assembly step requires that the 360 system macro library and the SOUPAC macro library be concatenated. At the University of Illinois, the monitor is linkedited into the data set SYS1.LINKUOI. Since this is a "protected" data set, completion of the linkedit step requires operator console intervention.

4. Create the program library. At the University of Illinois this library is named SYS4.SOUPLIB. Because of the size of the program library, each program is added individually. Addition of a program requires an assembly, a FORTRAN compilation and a linkedit. The assembly requires that the 360 system macro library and the SOUPAC macro library be concatenated. The linkedit step requires

the concatenation of the SOUPAC subroutine library, the IBM library SYS1.FORTLIB, and the University of Illinois library SYS1.FORTUOI. Member names is SYS4.SOUPLIB must correspond to the table entries in the subroutine MAINS in the Syntax Interpreter. In addition, the Syntax Interpreter must have the member name SEARCH, and the special exit routine (see section VI.A) must have the member name EXIT.

5. Create cataloged procedures for ease in invoking the system. Refer to section II.C for minimum JCL requirements of the SOUPAC system.

6. If it is desired to maintain log counts of each job step and problem program run, create a log data set. This data set should have 48 four-byte words per logical and physical record. There should be one record for each problem program and five records for the Syntax Interpreter. At the University of Illinois this data set is named SYS4.SOUPLOG.

Before creating the SOUPAC system, the decision must be made whether or not a SOUPLOG data set will be maintained. Two assemblies are dependent upon this decision; the assembly of subroutine TIOT in the Syntax Interpreter, and the assembly of subroutine NEXT in the SOUPAC subroutine library. Both TIOT and NEXT are conditional assemblies (i.e. macros) which require the argument ACTIVITY if a SOUPLOG data set is to be maintained. If a SOUPLOG data set is not to be maintained, ACTIVITY should not be coded as an argument to either the TIOT or NEXT conditional assemblies, and it is not necessary to perform step 6 above or to include a SOUPLOG ddcard in any cataloged procedures created in step 5 above.

II. The Executive Monitor

A. The loader

The loader, with csect name UISOUPAC, receives initial control whenever the SOUPAC system is executed. To make the rest of the system available to itself, the loader opens a DCB for the ddname SOUPLIB which points to the SOUPAC program library. All SOUPAC programs are contained in the program library and are executable load modules (i.e. have been link-edited with all the right subroutine libraries) and are ready to be invoked by the loader. When a particular program is to be invoked, the member name of the load module must be in location MEMBER. To invoke the program, the loader branches to CALLPGM. A BLDL macro is executed for the name in MEMBER in order to set up the information necessary for the actual link. Upon successful completion of the BLDL, a LINK is executed and control goes to the assembly language MAIN of the desired program.

The first program linked to is the Syntax Interpreter which has the member name SEARCH. The name SEARCH is always in MEMBER whenever UISOUPAC is initially brought into memory. Upon return from SEARCH, there will be in location PAUL a count of the number of programs which the loader is to invoke. The program count is tested for non-positive. This much is always done for each SOUPAC job step. If the program count is non-positive, control goes to location ENDTASK, the normal exit from UISOUPAC. If the program count is positive, location ANYSNAPS is checked to see if SEARCH indicated that snapdumps had been requested. If snapdumps were requested, two data sets are opened, SNAPDUMP for snapdump output and PGMQUEUE for input of the loading queue. If snapdumps were not requested (i.e. if ANYSNAPS is zero), only PGMQUEUE is opened.

The data set pointed to by the ddname PGMQUEUE contains the loading queue, created by SEARCH, in twenty byte logical records. For each such record, one problem program is invoked. As each program successfully finishes, the program count in PAUL is decreased by one, and the next program in the queue is invoked. This process continues until the program count goes to zero (i.e. the loading queue is exhausted), or until an error occurs. If no error occurs before the queue is exhausted, control falls through to ENDTASK and the loader takes a normal exit.

Each twenty byte record contains the eight byte member name of the program desired and is followed by a four byte snapflag, a four byte memory allocation request, and a four byte integer record number. The snapflag is used by subroutine NEXT during termination of a program to determine what type of snapdump, if any, was requested. The memory allocation request is used by problem programs which do dynamic storage allocation and indicates how much storage should be reserved for system use during running of the problem program. The low order two bytes of the four byte record number are moved into the low order two bytes of APGM (see section III.B). This record number is used by subroutine NEXT for writing out activity information to SOUPLOG on the current program.

All programs carry one of three possible return codes upon returning to the loader. This return code is placed in location CODE by subroutine NEXT. A return code 8 indicates normal termination of the program via CALL NEXT and permits the loader to continue processing the loading queue. A return code 12 indicates an error termination via CALL ERROR. A return code 16 indicates that an execution error was processed by the FORTRAN extended error monitor facility, and that execution was terminated by a call from level one of the program's MAIN to FORTERR (see section III.A). Return codes 12 and 16 branch to ERRETURN and

TRACEBCK respectively to print out which of the two error types occurred. Both types then proceed to print out additional diagnostic information and finally to terminate the job step. Diagnostic information consists of the address of the entry point, the address of blank common, and the member name of the current program, the number of the current data deck, the number of the last card in the current data deck, and the image of the last card read. The address of blank common is provided as a debugging aid for finding program parameters in a dump since the main parameters are stored in the first 96 locations of blank common in an array conventionally named IPAR. The number of the last card read is the result of a physical count of card images in the current data deck and includes all data format cards and the END# card if it was the last card read.

The SOUPAC loader can terminate either normally or in one of three error exits. The first concern, no matter what way the loader terminates, is to make sure that all FIOCS# and DIOCS# controlled data sets are closed. A necessary condition to closing the data sets is that some copy of IBCOM# be available. Normal termination of the loader utilizes the fact that all programs in the SOUPAC library, and in particular the last program invoked, contain a copy of IBCOM#. When it is known that the current program, either a problem program or the Syntax Interpreter, is the last program to be invoked, location ENDFLAG is set to one. If the last program invoked is the Syntax Interpreter, the Syntax Interpreter itself will set ENDFLAG to one. If the last program to be invoked is any other program, the loader will set ENDFLAG to one when the program count in PAUL has been found to be one. When ENDFLAG is one, subroutine NEXT places the address of entry point PGMEND into ENDFLAG and returns directly to the loader instead of to the supervisor (see section IV.B).

Returning directly to the loader ensures that IBCOM# will still be available when control in the loader goes to ENDTASK. ENDTASK sets location CODE to four and calls CLOSEIO1 which closes all FIOCS# and DIOCS# controlled data sets. The loader must now return to the supervisor twice, first to deallocate the last program, and second to terminate the job step. Therefore, the loader saves the address of PGMEND from ENDFLAG, sets ENDFLAG to two, and returns to the supervisor using the savearea pointed to by PGMEND. This deallocates the last program. When the loader regains control from the supervisor, ENDFLAG is now sensed to equal two, and control goes directly to location AMEN where the final coup de grace of normal termination is accomplished.

Error cases one and two, resulting eventually in completion codes 12 and 16 respectively, are identical except for the message each prints out. Both return directly to the loader as is done in normal termination instead of to the supervisor. This has the double advantage of keeping IBCOM# available for data set closings and of having the program still around for an abend dump, if desired. When the loader gains control from either error condition, CLOSEIO1 is called. Both error conditions print out what they have to say and then call QUIT.

Error case three, error while in the loader, is a general category which is handled differently depending upon the particular faux pas. If the loader is unable to open SNAPDUMP or PGMQUEUE, if it is unable to complete the BLDL for any program other than the special program EXIT, or if an end of file condition is reached on PGMQUEUE, CLOSEIO2 is called and eventually QUIT is called. Calling CLOSEIO2 causes EXIT, which contains a copy of IBCOM#, to be invoked. EXIT closes the data sets controlled by FIOCS# and DIOCS# and returns control to the loader. If the loader is unable to open SOUPLIB or FT06F001, if a program

returns an invalid return code, or if BLDL fails for the error-in-monitor routine EXIT, QUIT is called immediately. By far the most common error which occurs while control is in the monitor is the failure of BLDL. Typically the member name which cannot be found will be printed out along with all information concerning the last card read by the system. This particular error will result in a completion code 20. In all other cases of error while in the monitor, the completion code will be whatever happens to be in location CODE from the last time it was set.

Final termination for all error conditions is by a "BAL 10,QUIT" instruction. Use of this instruction is standard throughout the loader so that whenever any error condition causes termination, register 10 will contain the address following the BAL instruction which requested termination. Calling QUIT closes the four loader controlled data sets and results in an ABEND supervisor call with the user completion code being whatever is left in location CODE.

Note the internal service subroutine EBCDICVT. This routine is used to convert the contents of register 9 to a base 16 integer character string representation for printout. EBCDICVT is called for setting up address constants to be printed out in the error diagnostics. Identical code can be found in subroutine NEXT for printing out address constants when SNAPDUMPS are executed.

The table TABLE1 is a set of address constants of the entry points to the I/O control section of the monitor. TABLE1, pointed by TAB in the SVT, is copied by the MAIN of each program into the entry point XADDR in subroutine INTRFACE. It is this table which provides to INTRFACE the mechanism by which the problem program references the I/O control section of the monitor.

B. The I/O monitor

SOUPAC performs all input of card images, output of line images, output of punched card images, and the I/O of user-addressable data sets using the IBM FORTRAN subroutine library. Both sequential and direct access I/O are supported by SOUPAC. SOUPAC convention is to consider all data sets with Data Set Reference Numbers (DSRN's) 1 through 50 as sequential files and all data sets with DSRN's 51 through 99 as direct access files. The first ten sequential files are reserved for use by the SOUPAC system (DSRN's 1 through 10). The next forty sequential files are addressable by the SOUPAC user as S1 through S40 (DSRN's 11 through 50). The remaining forty-nine files are useable as direct access files only and are addressable by the SOUPAC user as D1 through D49 (DSRN's 51 through 99). Note that D49 should be avoided as a user address since it is this data set (i.e. FT99F001) which is used by certain programs for direct access roll memory (see section IV.F).

The I/O routines of the FORTRAN subroutine library are in two basic levels. The first level is represented by IBCOM#. The second level is represented by FIOCS# and DIOCS#. All read and write statements result in one or more calls to IBCOM#. IBCOM# serves as an intermediary between the problem program code and the second level routines which execute the actual I/O interrupts. IBCOM# receives buffer pointers from FIOCS# and DIOCS# and moves data between the I/O buffers and program variables with or without format conversion as required. IBCOM# also serves a myriad of other supervisory functions over the running problem program code. FIOCS# and DIOCS# get buffer space and manage buffer pointers, watch over the DCB's for the data sets, and issue the I/O interrupts.

These levels are split up within the SOUPAC system so that each program load module has its own copy of IBCOM# and associated level one

routines, while all level two routines are contained only within the I/O monitor. INTRFACE, contained in the SOUPAC subroutine library and linkedited into all modules in the program library, is used to connect the two levels at run time. Whenever any program wants to call a routine in the I/O monitor, the program performs a call to the entry point in INTRFACE which has the same name as the desired I/O monitor routine.

The ability to cross reference between INTRFACE and the I/O monitor routines is established by the assembly language MAIN at the beginning of execution of each program (see section III.A). One of the major functions of MAIN is the shifting of address constants between the monitor and the program's copy of INTRFACE. One of the criterions for deciding whether a particular control section of the IBM FORTRAN subroutine library was to be included in the I/O monitor or not was to include in the I/O monitor that code which would minimize the amount of address constant shifting that MAIN would be required to perform. The shifting of address constants from the monitor to the program is a straightforward copying of TABLE1 in the loader to entry XADDR in INTRFACE. Establishing addressability in the other direction, however, involves copying individual address constants into specific locations within the code of the I/O monitor routines. With different releases of the IBM FORTRAN subroutine library, the offsets of these specific locations relative to the beginning of each affected I/O monitor routine may vary. To the extent that the offsets vary, the MAIN macro is clearly release dependent.

The following is a rundown of the control sections within the I/O monitor:

FIOCS# is the IBM FORTRAN subroutine library I/O interface to BSAM.

FIOCS# handles all bookkeeping of memory requirements for DCB's and I/O buffers. FIOCS# provides to its calling program pointer information for the next logical record to be written or read. FIOCS# controls the actual opens and closes for all standard FORTRAN data sets.

DIOCS# is the direct access method counterpart to FIOCS#. DIOCS# provides an interface to BDAM. DIOCS# executes all define files.

IHCERRM is the IBM FORTRAN subroutine library extended error monitor facility.

IHCUATBL is the IBM FORTRAN subroutine library Unit Assignment Table.

SOUPDADS requests DIOCS# to execute any define files needed for a particular SOUPAC run. Entry at DABTBL is used to build the appropriate tables necessary for define files to be properly executed. DABTBL is called once for each #DEFINE control card found in the prolog of a SOUPAC user's program. After all table additions have been completed, a call to DADSET scans the tables and executes a define file for each data set which has appropriate table entries. DABTBL and DADSET are called only from the Syntax Interpreter. DADSET is called once and only once, from subroutine MAINS of the Syntax Interpreter, per SOUPAC run.

OUTSET is used to save the value of the number of rows (i.e. number of logical records) for each sequential file used for temporary storage. This information is used most importantly by ROWIN.

DUMMY01 is a csect whose sole purpose is to eliminate unresolved external references which would otherwise exist within the I/O monitor because of the separation of the I/O code into two levels.

LOGIN is the subroutine which reads a particular record from the data set pointed to by the ddname SOUPLOG. Entry LOGOUT is called to write back the record read by the previous call to LOGIN. Reading and writing of SOUPLOG is done by BDAM. An ENQ macro instruction is executed during call to LOGIN to prevent more than one running SOUPAC job step from accessing SOUPLOG at a time. A call to LOGOUT executes a DEQ macro instruction after the record has been written back onto the data set. LOGIN receives a record number in register 1 from LOGIN's calling program. This record number is decreased by 1 so that the first record in the file is addressed as record 0, etc. Location FLAG is checked to see if the data set has been opened, and if the data set has not yet been opened, an OPEN is executed. LOGIN returns to its calling program in register 1 the address of the buffer into which the data record has been read.

C. JCL requirements

In order to execute a SOUPAC job step, certain JCL requirements must be observed. To invoke the SOUPAC system, a "// EXEC PGM=UISOUPAC" card must appear. This card causes the monitor to be loaded and given control. At a minimum, the following DD cards must appear.

SOUPLIB: The program library.

PGMQUEUE: The loading queue. The loading queue is created by the Syntax Interpreter and is used by the loader, UISOUPAC, to determine which members from the program library are to be run.

FT03F001: The problem program parameter file. This file is created by the Syntax Interpreter and is referenced by each problem program.

FT05F001: Card image input file.

FT06F001: Line printer image output file.

SOUPLOG: Direct access program activity file.

Certain individual programs require the use of FT01F001 and FT02F001 as internal scratch data sets not accessible to the general user. If card image output (punch) is desired, an FT07F001 card is required. Programs which perform dynamic, run time memory allocation require an FT99F001 card to provide a data set for roll memory.

For each user addressable data set to be used, a corresponding DD card must be included. All such DD cards have ddnames of the FORTRAN form FTxxFyyy where "xx" is the data set reference number (DSRN) and "yyy" is the file number. For SOUPAC addresses S1 through S40, "xx" is 11 through 50. For SOUPAC addresses D1 through D49, "xx" is 51 through 99. File numbers start at 001 and increase by one for each data set associated with the DSRN.

A. MAIN

The MAIN macro instruction constitutes the assembly language program, with csect name MAIN, which receives control from the supervisor as a result of the SOUPAC loader executing a link SVC. MAIN is used as the front end control section for all programs in the SOUPAC program library. The MAIN macro has five parameters in the operand field. The first parameter is the name of the subroutine which MAIN is to call to begin execution of the problem program. The second parameter is the date in eight or fewer characters. The third parameter is a character string, enclosed in apostrophes, which is used by MAIN and by subroutine TYME for printing out the program name. The fourth parameter is the optional word YES. This parameter indicates that subroutine ALLOC8 is to be called. ALLOC8 performs a GETMAIN for programs which perform run time storage allocation (see section III.C). The fifth parameter is the optional word NO. This parameter indicates that no calls to ROWIN, ROWOUT, TWOIN, or TWOOUT exist in the problem program. For example, this parameter is currently used in Linear Programming. Without NO being coded as a fifth parameter, a program such as L-P would have DECKADDR as an unresolved external reference when being linkedited.

MAIN is coded in two levels, the first level "calling" the second. The reason for two levels is that in the second level, a call to the initialization section of IBCOM# is made. During this initialization register 13 is saved by IBCOM# so that in case the FORTRAN error handling facility finds an error, it can get the savearea for the program which performs the initialization and return to that program's caller. In a normal FORTRAN program (i.e. not a SOUPAC program), the main calls the

initialization routine immediately, so that under appropriate error conditions, any further execution of the problem program is bypassed and control goes directly to the program's caller (i.e. usually the supervisor). In the case of SOUPAC, however, we do not want to return to the calling program (the supervisor) directly, but rather we want MAIN to regain control after such errors. By faking out the FORTLIB error monitor, control goes to level two's "caller", level one, which then calls entry FORTERR in subroutine NEXT. This call is the only place in each program loadmodule which calls FORTERR and also is the process by which a return code 16 is generated (see section IV.B).

The initialization call to IBCOM# serves two other basic functions. First, the address of the entry point to the program (in SOUPAC'S case the will be the address of LEVEL2) is saved so that in printing out a savearea traceback for errors, the error monitor can compare the stored value for register 15 in each savearea against the entry point to the program and know when to stop the traceback. Second, a SPIE SVC is executed to give the FORTLIB interrupt handler, ARITH#, control after selected program interrupts.

Level one sets up the following address constants:

1. Into the first word at entry PGMEND in subroutine NEXT goes the address of the savearea for level one so that NEXT can return properly to the supervisor upon normal termination of the load module. Into the second word at PGMEND goes the address of the SVT so that NEXT can reference the SVT correctly.
2. Into the first two locations of the SVT goes the address of blank common and the true entry point of the load module.
3. Into the table beginning at entry XADDR in INTRFACE goes the list of all adcons necessary for the program to reference the I/O monitor in UISOUPAC.

4. Into the routines in the I/O monitor of UISOUPAC go all adcons necessary for the I/O monitor to reference needed control sections in the problem program. Into FDIOCS# entry in IBCOM# goes the address of the special entry to DIOCS#, IBCENTRY, for performing direct access I/O.

No handy interface was written for the I/O monitor to reference the problem program in an analogous manner to the way INTRFACE references the I/O monitor because references from INTRFACE to the I/O monitor are all straightforward calls, while references from the I/O monitor to the problem program are not branches but table references or other non-standardized practices.

Level two performs the following functions:

1. Execution of the initialization call to IBCOM#.
2. Initialize execution timing of the load module by a STIMER macro.
3. Read in the main program parameters into IPAR (the beginning of blank common).
4. Print out the number within the loading queue of the current program and also print out the name of the program. MAIN computes the number within the loading queue by adding one to PGMCOUNT in the SVT. The name of the program comes from the third argument to the MAIN macro.
5. Calls entry RCOPY in subroutine READ.
6. Stores the address of DECKNO of the SVT into DECKADDR of subroutine FREAD1.
7. Copies the first four words of ADCON# into entry CTAB0 in BCNVT and copies the table CTAB1 into the first four words of ADCON#. CTAB1 contains addresses for entry to BCNVT corresponding to the format conversion routines in ADCON# for F format input, F format

output, E format input, and E format output.

8. Calls ALLOC8 and prints the starting address and the length of the memory block getmained by ALLOC8.
9. Calls the program.

Functions 3 thru 6 are not performed for the Syntax Interpreter. Functions 5 and 6 are not performed for problem programs which have NO coded as the fifth argument on the MAIN macro card. Function 8 is performed only for those programs which have yes coded as the fourth parameter on the MAIN macro card (i.e. those programs which do run time storage allocation).

Entry TYMEARG is used by subroutine NEXT as the address table of arguments for calling subroutine TYME. TYME needs to know when MAIN initiated execution timing and also needs to know the program name. NEXT does not have this information whereas MAIN does, therefore NEXT uses TYMEARG to provide the arguments to TYME.

Problem programs which return to MAIN instead of executing a CALL NEXT get a call to NEXT executed for them.

B. SVT (SOUPAC Vector Table)

The SVT macro provides a convenient way for SOUPAC assembly language code to reference symbolically any location in the SOUPAC Vector Table. The SVT macro has one argument which is the word DUMMY. Coding the word DUMMY causes a DSECT to be generated while not coding DUMMY causes the actual table to be created. All uses of SVT have the word DUMMY coded except for the one use in UISOUPAC where the actual table is desired. The purpose of the SVT is to provide a means of communication between the problem program and the loader. Each problem program contains in Register 1 the address of the actual SVT at the time assembly language MAIN receives control from the supervisor.

The SOUPAC Vector table contains the following information:

ACOMMON: the address of blank common in the current program. This address is entered by MAIN of each program.

EP: the address of the entry point MAIN in the current program. This address is entered by MAIN of each program.

TAB: the address of the table of I/O monitor routine addresses. This table of addresses is needed by MAIN to make the I/O monitor addressable by INTRFACE in each problem program.

AERRSAVE: the address of ERRSAVE. ERRSAVE is a special savearea set up so that when a program terminates execution by a CALL ERROR or a call to FORTERR (from MAIN), the program can return directly to UISOUPAC instead of going through the supervisor. This is done so that the supervisor won't de-allocate the space occupied by the problem program. If the space were de-allocated, it could not appear in a dump upon the abend.

PAUL: This location serves two purposes. PAUL is used initially to pass to SEARCH the address of the parameter field from the EXEC statement. SEARCH uses this address to examine the parameter field and determine the "SOUPAC OPTIONS" for the task. SEARCH then uses PAUL to store the count of the number of programs in the execution queue which the loader is to invoke.

ABNDCODE and CODE: this location contains the current SOUPAC completion code.

ATLOC: the address following the call to NEXT, ERROR, or FORTERR which terminated the current problem program. This address is entered by NEXT of each program.

SNAPFLAG: the snapflag used to indicate what type of snapdump was requested. The snapflag is saved from the last four bytes of each loading queue logical record. Snapflag is eventually checked by NEXT at the termination of execution of the problem program.

ANYSNAPS: the location where SEARCH stores the number of snapdumps to be executed. If this location is non-zero, the loader attempts to open a data set for snapdump output. If the open is successful, the loader places the address of the open snapdump DCB in this location. If the open is unsuccessful, the task is terminated.

DECKNO: the count of card input data decks handled. The address of DECKNO is placed in subroutine FREAD1 by MAIN for all those programs which do not have the fourth parameter of MAIN coded as NO. When an error termination occurs before FREAD1 finds a

DATA format card, DECKNO is still zero and the last card read will have been read by SEARCH. Hence, the user program deck, all cards up to and including the ENDS card, will constitute deck number zero in the error diagnostics printed by UISOUPAC for such an error termination.

PGMCOUNT: the number in packed decimal of the current program within the loading queue. This location is increased by one by the MAIN of each program except SEARCH.

TOTALPGM: the total number of programs in the loading queue; the number of records in the loading queue. This information is not used anywhere but is included for completeness in the case of a dump.

AIMAGE: the address of an 80 byte buffer into which subroutine NEXT copies the image of the last card read by the current program if, in fact, the current program read any cards.

NCARD: If the last card read was not an END# card, this location contains the number of that card within the current data deck. DATA format cards are included in this count. If the last card read was an END# card, this location contains zero. NCARD is set by NEXT from information contained in INTRFACE.

NEND#: If the last card read was an END# card, this location contains the number of the END# card within the current data deck. DATA format cards are included in this count. This location is set by NEXT from information contained in INTRFACE.

ENDFLAG: This location is zero except if the current program is the last one the loader is to invoke. Whenever NEXT returns directly to the loader instead of to the supervisor, NEXT

places in ENDFLAG the address of entry PGMEND. ENDFLAG is set to two by the loader immediately prior to its returning to the supervisor for deallocation of the last program invoked. ENDFLAG is set to three by the loader whenever CLOSEIO2 is called (i.e. whenever the special program EXIT is invoked).

MEMORY: the amount of memory, in bytes, which the current program needs to leave for system use. MEMORY is used by only those programs which do run time storage allocation (see section III.C). This location is filled by the loader from the last four bytes of the current loading queue record.

APGM: the address of the DCB for SOUPLIB. APGM is used by the Syntax Interpreter to reference the program library when determining run time memory requirements. APGM is also used through the remainder of the SOUPAC run to pass subroutine NEXT information for writing on the program activity data set, SOUPLOG. APGM contains the record number to be used for the current program in the low order halfword, and contains a value to be used as a month index register stored in the high order halfword.

C. ALLOC8

The ALLOC8 macro is used to perform the GETMAIN for all SOUPAC library programs which perform run time storage allocation. ALLOC8 is called by MAIN whenever the MAIN macro has YES coded as the fourth macro argument (see section III.A). The ALLOC8 macro has one argument, an integer which represents the minimum number of bytes the problem program needs for storage space in order to run. This integer must be a multiple of eight.

ALLOC8 performs a variable-conditional (VC) type GETMAIN for as much contiguous memory as is available. The amount of memory returned by the supervisor is checked against a minimum amount required. This minimum requirement is the sum of the minimum storage required by the problem program as expressed by the macro argument plus the amount of memory to be returned to the supervisor for system use during the running of the problem program. If the amount secured by the GETMAIN is less than the sum of program plus supervisor memory required, control goes to NOROOM. If, however, sufficient space is available, the top end of the secured memory block is immediately returned to the system via a FREEMAIN. The length returned is the amount specified in MEMORY in the SVT (see Section III.B). The value for MEMORY is created by the Syntax Interpreter and is passed thru PGMQUEUE (see Section II.A).

ALLOC8 stores the address of CBLOCK into eight past entry PGMEND in subroutine NEXT. NEXT uses CBLOCK to perform a FREEMAIN (see section IV.B). Finally, ALLOC8 sets up the proper calling arguments for MAIN to use in calling the problem program, zeros out the block of memory to be used by the problem program, and then returns to MAIN.

Notice that ALLOC8 uses SVTREG to point to the SVT, but the register is never initialized within ALLOC8. The reason for this is that ALLOC8 is always called from MAIN, and MAIN uses the same register to point to the SVT.

D. RECALL

RECALL has an indefinite number of arguments which are used as control sections to be called. For each csect name listed as a macro argument, a call is made to that control section. When RECALL is called, general register 1 points to the table of argument addresses which are to be passed to all the listed control sections. The purpose of RECALL is to get this list of arguments broadcast to all the listed control sections and to provide to all programs of a load module the information necessary to reference arrays created from the memory block secured by ALLOC8.

IV. The SOUPAC Subroutine Library
- System Subroutines -

A. INTRFACE *

INTRFACE is the key link in establishing communication between a program and the I/O monitor (see sections II.B and III.A). All calls from a program to routines in the I/O monitor are accomplished by calling the entry point in INTRFACE which has the same name as the desired I/O monitor routine. All such entries in INTRFACE, except those for FIOCS# and DIOCS#, simply load the "branching register" with the address of the for-real routine in the I/O monitor and branch to that address. Calls to DIOCS# result in a call to ERROR since a call to DIOCS# is a define file, and no program should be executing a define file directly. All define files are handled by the Syntax Interpreter thru the routines DADSET and DABTBL.

The interface for FIOCS# is not so simple, however. Advantage is taken of the fact that all sequential I/O must pass thru INTRFACE on its way to the real FIOCS#. In particular, SOUPAC peeks at all card images coming in (i.e. all logical records from the DSRN 5 input buffer) and at all line printer images going out (i.e. all logical records to the DSRN 6 output buffer). The reason for looking at card images coming thru for DSRN 5 is to find END# cards in the input stream. An END# card is defined to be a card with the characters END# as the first four non-blank characters on a card. Whenever an END# card is found, INTRFACE simulates an end-of-file return to the statement indicated by

---

* The initial inspiration for INTRFACE grew out of a subroutine called FIO999 written by John R. Ehrman in October 1966 for the Stanford Linear Accelerator Center (SLAC) Computation Group. See Appendix A.

the END= option of the read statement. The reason for looking at the line printer output images for DSRN 6 is to count lines to the printer and every sixty lines insert a '1' as carriage control for page ejection. This prevents SOUPAC programs from printing over the page separations.

Several additional related functions are performed by INTRFACE for card image input from DSRN 5. As each image is read, it is saved in an 80-byte buffer called LASTCARD. Later, as each program terminates, NEXT checks to see if any cards were read during the execution of the program (i.e. location RECPOINT is non-zero). If any cards were read, NEXT copies the contents of LASTCARD into the loader in a buffer called IMAGE. This enables the loader to print out in the error diagnostics an image of the last card read. At the same time INTRFACE keeps in NCARDS a count of the number of card images read per data deck. Whenever an END# card is found, the number of the END# card is saved in END#CARD, and NCARDS is reset to zero. This information is similarly shifted into the loader by NEXT and is used in the error diagnostics (see sections II.A and IV.B).

Also, a count of the card images read per FORTRAN read statement is kept in IPAR(72). The count is possible because for any read statement, the first image is requested by an init request, and subsequent images for that statement are requested by read requests (see below about init and read requests to FIOCS#); on an init request the count is set to one, and on a read request the count is increased by one. IPAR(72) is used by subroutine READ to make sure that each time a FORTRAN read is executed for a given data deck, the same number of cards are requested. In particular, the last FORTRAN read executed for a data deck may come up short if cards are missing from the deck (see section IV.D.2). Whenever an END# card is found, IPAR(72) is set negative so that subroutine READ can know if the end-of-file condition resulted from an END# card or an actual end-of-file.

Besides counting the number of card images requested per FORTRAN

read statement, INTRFACE also saves all the incoming card images in a special buffer up to the capacity of the buffer. This facility is invoked by a call to the entry point COPY in INTRFACE. Note that when MAIN of a problem program is entered, one of the functions performed is a call to entry RCOPY in subroutine READ (see sections III.A and IV.D.2). RCOPY calls COPY and provides to INTRFACE the address and the length in bytes of the special buffer, called CBUF, into which the card images are to be stored. As the execution of each FORTRAN read from DSRN 5 progresses, INTRFACE saves the successive images in CBUF. As each read completes, subroutine READ checks IPAR(30) for the value "true" to see if the user included a #COPY control card immediately preceding the current problem program. If one was included, then READ prints out the contents of CBUF so that the user gets a card image listing of his data deck grouped by logical rows. If there are more cards per logical row than CBUF can save, the excess images are not saved. If the user has indicated that he wants his input deck checked for proper sequencing, READ calls subroutine SEQCHK and SEQCHK looks at the images in CBUF for valid sequencing.

Note that the saving of card images in CBUF is independent of the use of LASTCARD. LASTCARD holds the most recent card image read whether it was also saved in CBUF or not. CBUF saves at one time all the card images from the execution of a given read from DSRN 5 up to the maximum which CBUF can hold.

INTRFACE saves the buffer pointer provided by RCOPY (READ). As each init request for input from DSRN 5 comes thru for FIOCS# (i.e. each time the FORTRAN read statement is executed afresh), the pointer is reset to the initial address provided to COPY. As each card image comes back from FIOCS#, the pointer is increased by 80 so that any additional card image requests to FIOCS# before the next init will have the card image stored into the next available 80-byte block of CBUF.

There are five options to FIOCS#; init, read, write, control, and

terminate. INTRFACE is concerned only with init, read, and write requests, and is concerned with such requests only for DSRN's 5 and 6. Init always knows what the DSRN is, whereas read and write requests do not. Since read and write requests are always preceded by an init request, init turns location FLAG on whenever the init is for DSRN 5 or for DSRN 6. In this manner, subsequent read or write requests are either acted upon by INTRFACE or are simply passed along to FIOCS# without further action depending upon the status of FLAG. It is assumed that there is no attempt to read from DSRN 6 or to write to DSRN 5.

As a call to FIOCS# comes thru INTRFACE, the option type is checked. If a control or terminate type request is being made, INTRFACE turns FLAG off and proceeds to call the real FIOCS#. If an init request is being made, the DSRN is checked for being either 5 or 6. If the DSRN is neither, INTRFACE proceeds to the same code sequence used by the control and terminate options. If the DSRN is 5, control goes to READCHK; if the DSRN is 6, control goes to WRITECHK. At both READCHK and WRITECHK, FLAG is turned on. If the option type indicates that a read or write request is being made, FLAG is checked to see if it was turned on by a previous init. If FLAG is off, the last init was not for DSRN 5 (for read requests) or for DSRN 6 (for write requests), and INTRFACE proceeds to the same code sequence used by the control and terminate options. If FLAG is on, read or write remains in control.

The action taken by INTRFACE when control goes to READCHK (on an init for input from DSRN 5) and for any subsequent read requests is essentially the same. The single difference is that for READCHK, the pointer which is maintained for storing consecutive card images in the special buffer provided by COPY's caller is reset to point to the beginning

of the buffer. Both READCHK and the read request when FLAG is on branch to location GETPOINT. This calls the real FIOCS# and returns to INTRFACE at RUNTEST with a pointer in register 2 to the beginning of the next logical record. Register 3 contains the length of the logical record, which for normal card input is assumed to be always 80 bytes. Upon return from FIOCS#, registers 2 and 3 are saved in RECPOINT; the card image is copied into LASTCARD; the card count of the number of logical records per FORTRAN read statement, kept in IPAR(72) and referred to by INTRFACE as CARDREC, is increased by one; and the count in NCARDS, the number of cards read per data deck, is increased by one. The pointer to the COPY buffer is checked for zero. If the pointer is zero, COPY was never called, and the code to save the new card image in the COPY buffer is skipped. Note that if COPY has been called, consecutive card images are saved as each read request comes thru until it is determined that any additional card images will not fit in the buffer.

The card image is next checked for the characters END# as the first four non-blank characters (i.e. blanks are ignored). If the image is not of an END# card, INTRFACE returns to its caller, IBCOM#, normally; otherwise INTRFACE branches to EOF. At EOF the value in NCARDS is copied into END#CARD so that the count of the END# card can be kept while NCARDS is reset to zero for the next data deck which might come by. IPAR(40) is set negative to indicate that an END# card has indeed been found. INTRFACE then simulates an end-of-file return to the address specified, and pointed to by an address constant saved in IBCOM#, in the END= option of the FORTRAN read statement.

WRITECHK (i.e. an init request for output to DSRN 6), after turning FLAG on, calls FIOCS# to get buffer pointers to a buffer where INTRFACE's

caller can put its next print line. Upon return from FIOCS# at location FIOWINIT, INTRFACE merely saves the buffer pointers in OUTPOINT. No checking of the output buffer is done because nothing is in it yet except for blanks put there by FIOCS# to initialize the buffer. On subsequent write requests, the carriage control character of the line image is checked. The line image is now in the buffer because it was placed there by INTRFACE's caller just before the write request was issued. If the carriage control character already is a '1', INTRFACE branches to PAGESET+4 to reset the counter in location LINESOUT to one. If the carriage control is other than a '1', it is checked, in order, for being either '+', '<blank>', '0', or '-' resulting in an addition to LINESOUT of 0, 1, 2, or 3 respectively. If the carriage control character is other than one of the five checked, it is set to a blank and one is added to LINESOUT. If LINESOUT is now greater than 60, the carriage control character is set to '1', and LINESOUT is reset to one. INTRFACE then branches to GETPOINT which calls FIOCS# to get pointers to the next available output buffer. Again, upon return from FIOCS#, the pointers are saved in OUTPOINT. INTRFACE then returns to its caller.

Note that for a FORTRAN read statement, "n" logical records can be read by an init and "n-1" read requests. If only one logical record is to be read, the whole read statement is done by the one init request and no read requests to FIOCS# are needed. For a FORTRAN write statement to write "n" logical records, however, an init is needed just to get the first set of buffer pointers, and "n" write requests are needed to transmit the "n" logical records. To do just one logical record, both an init and a write request are required. Each init request always returns to INTRFACE's caller with pointers to the next buffer, but an init for

input has the first record in the buffer ready for it, while the init for output requires an additional write request to inform FIOCS# that the buffer pointed to after the init has been filled.

All calls to the I/O monitor thru INTRFACE other than init, read and write requests to FIOCS# for DSRN's 5 and 6 return directly to INTRFACE's caller and do not go thru INTRFACE when returning from the I/O monitor.

B. NEXT

A call to subroutine NEXT, or either of the two entry points ERROR and FORTERR, causes termination of the problem program. Entry via a CALL NEXT results in a completion code 8 being set in location CODE of the SVT. Calls to ERROR or FORTERR result in a completion code 12 or 16 respectively. All three entry points cause control to return to the loader either directly or through the supervisor. Returning through the supervisor deallocates the load module first, and then returns control to the loader. If NEXT is called and ENDFLAG in the SVT is zero, control returns through the supervisor. If either ERROR or FORTERR are called, or if NEXT is called while ENDFLAG is non-zero, control is returned directly to the loader (see section II.A). Before returning control to the supervisor, the subroutine performs snapdumps if requested, and calls TYME which prints out the execution time for the problem program. The subroutine also checks to see if any card images were read from DSRN 5. If card images were read, the last one read is copied from LASTCARD, an entry in INTRFACE, into IMAGE, a buffer in the loader; the number of the last card read within the current data deck is copied into the SVT. If NEXT was called and if a GETMAIN was performed by ALLOC8 for the problem program, a FREEMAIN is executed.

When any one of NEXT, ERROR, or FORTERR is called, the completion code is immediately set and control goes to location ORDER. The subroutine gets the address of the SVT which was placed in location ASVT by MAIN when the problem program was invoked (see section III.A). The completion code is stored into the SVT. Register 14 is saved in ATLOC of the SVT so that the loader can print out in the error diagnostics the address where ERROR or FORTERR was called.

Before calling TYME, the first byte of the special savearea used by IBCOM# must be set to X'FF'. The reason for this is that if the program

has terminated because of a call to FORTERR as the result of an I/O error, the first byte of the savearea is known by IBCOM# to be zero. IBCOM# checks this byte to prevent initiation of another I/O request after the previous one went bad. Whenever IBCOM# handles an I/O request which successfully terminates, it sets this flag byte to X'FF'. NEXT must set the flag byte to X'FF' to enable TYME to print out the execution time of the program. NEXT checks to make sure that looping does not occur (i.e. the sequence: bad I/O to DSRN 6; CALL FORTERR; CALL TYME to print out time; bad I/O to DSRN 6; CALL FORTERR, ad infinitum) by making sure that it does not call TYME twice in a row. This test is performed by checking LOOPFLAG for zero and setting LOOPFLAG to non-zero prior to calling TYME.

After TYME has been called, NEXT records activity information for the current program on SOUPLOG. This data set is a direct access file with a unique record corresponding to each SOUPAC program in the program library. NEXT gets the record number for the current program from the low order halfword of APGM. Each record has twelve groups, one group for each month, of four fullwords. The first fullword is increased by one. This will accumulate the number of times a program is run each month. The second fullword is used to accumulate total CPU time in centiseconds for the month. The last two fullwords are used to accumulate the sum of the centiseconds squared. At the end of the month this enables the calculation of frequency of program usage, average time in centiseconds and the standard deviation of execution time in centiseconds. The month is indicated by the high order halfword of APGM and is used directly as an index register. Note that an initial index value of four implies December, eight implies November, etc.

After writing out the current program's activity record, a snapdump is executed if it has been requested by the user (i.e. a #SNAP card was included in the user's program deck). The ID number of the snapdump is set from

location PGMCOUNT in the SVT. Note that after executing the snapdump, NEXT prints out entry point and blank common information identical to the message which appears in the error diagnostics printed by the loader prior to abend. NEXT also contains an identical copy of the "address of hex character string" conversion routine EBCDICVT which is contained in the loader. NEXT now calls FIOCS# to print out the line of asterisks which indicates the end of execution of the current program.

The last action which is taken for all three entries to NEXT is a check of entry RECPOINT in INTRFACE. If RECPOINT is zero, no cards were read by the current program. If RECPOINT is non-zero, cards were read and the image of the last card read, saved in the 80-byte buffer LASTCARD, is moved into the loader. NEXT uses AIMAGE in the SVT to provide the address of an 80-byte buffer in the loader for storing the image from LASTCARD. The number of the last card read within the current data deck is moved into the SVT (see sections III.B and IV.A).

At location TESTCC in NEXT, a test is performed to see if the program has been terminated by CALL NEXT or by a call to one of the two error entries. If the program terminated because of an error call, a branch is made to the exit at location SPECIAL. If the program terminated because of a CALL NEXT, two additional tests are made. First a test is made for location ACBLOCK. If ACBLOCK is non-zero, it contains the address of location CBLOCK within subroutine ALLOC8 (see section III.C). CBLOCK is a two word block which contains the starting address and length of an area for which NEXT executes a FREEMAIN macro instruction. If ACBLOCK in NEXT is zero, no FREEMAIN is executed. After the FREEMAIN is checked for and executed if necessary, NEXT makes a call on IOREW to rewind all the sequential data sets used during execution of the problem program. The last test performed in the case of

program termination via CALL NEXT is a test of location ENDFLAG in the SVT. If ENDFLAG is non-zero NEXT branches to location SPECIAL.

If the current program is the last program of the job step to be run, either because program termination is due to an error or because ENDFLAG is non-zero, control passes to location SPECIAL. SPECIAL returns directly to the loader without returning through the supervisor. Before returning, the address of PGMEND is stored into ENDFLAG in the SVT. The reason for saving the address of PGMEND is that the first word of PGMEND contains the address of the savearea passed to MAIN by the supervisor when the current (i.e. last) program was invoked. This savearea is used by the loader to return to the supervisor for deallocation of the last program under normal termination of the SOUPAC job step. NEXT gets the register information necessary to return directly to the loader for a fake savearea pointed to by AERRSAVE in the SVT (see section II.A).

If the program does not terminate by exiting via special, NEXT returns to the loader through the supervisor. The supervisor deallocates the program before giving control back to the loader. NEXT uses the savearea pointed to by PGMEND for returning to the supervisor.

Note that NEXT is set up as a conditional assembly (i.e. a macro) with one parameter. When the SOUPAC system is being generated, a choice must be made as to whether or not it is desired to record activity information on SOUPLOG. If activity information is desired, the statement used to invoke the NEXT macro must have the parameter ACTIVITY coded on the card. If the word ACTIVITY does not appear on the macro card, no activity information file processing for SOUPLOG is performed by NEXT.

C. TYME

Subroutine TYME is used to print out the CPU time elapsed during the execution of a program from the program library. TYME has three arguments; L, the time in centiseconds to be printed out; N, the length in bytes of the third argument IFORM; and IFORM, an array which contains the name of the current program. Subroutine TYME is always called from subroutine NEXT.

The name in IFORM is copied into the final print format array JFORM. The time is broken into two variables; J for seconds, and K for centiseconds. Appropriate variable formats in JFORM are set up for each value to be inserted into the print line so that no unnecessary blanks will occur in the final print line. The time message is finally printed out. TYME then returns to its caller.

D.0. The ROWIN complex

Within the SOUPAC system, all input of user data and all output of data to user controlled secondary storage, except for the few isolated instances noted below, is thru the set of routines described here as the ROWIN complex. The phrase "secondary storage" is emphasized because ROWIN does not perform printing or punching of output data. All ROWIN controlled I/O is either unformatted I/O to secondary storage or formatted card image input from DSRN 5. All unformatted I/O to user controlled secondary storage is done exclusively thru the ROWIN subroutine. Subroutines READ, FREAD1, CHKERR, and SEQCHK are used in conjunction with the ROWIN subroutine, but these two routines are concerned only with the input of card image data decks from DSRN 5. Subroutine IOREW, on the other hand, is not called from any of the other ROWIN subroutines but is rather called from NEXT. IOREW rewinds those sequential files which were used during execution of the program program.

The cases within SOUPAC where I/O operations are performed to secondary storage without going thru ROWIN are the INPUT, OUTPUT, FILE, and REWIND suboperations in the MATRIX program. The reason that INPUT and OUTPUT do not use ROWIN is that they each perform their own formatted I/O to a user data set. ROWIN's only provision for formatted I/O is restricted to input from DSRN 5. The reason that REWIND and FILE do not use ROWIN is that each performs an I/O operation which is not directly related to actual data record transmission.

It is essential for an understanding of ROWIN, indeed for an understanding of the data handling philosophy of SOUPAC, to recognize the fact that ROWIN handles only one row of data at a time. Each such row of data is treated as a single logical entity. ROWIN has no knowledge of its outside world except the parameter information passed on each separate call (i.e. ROWIN does not save information between calls). This is simply because

ROWIN can be requested, on successive calls, to perform either input or output to any unit address. This is the reason why ROWIN does not perform as much error checking as would be possible under more restrictive operating procedures. This is also the reason why ROWIN does not print or punch output since ROWIN does not have enough information available to itself to do the job correctly.

D.1 Subroutine ROWIN

The ROWIN subroutine is the primary means by which SOUPAC problem programs perform input and output of user data between primary and secondary storage. ROWIN also controls the input of card image data decks from DSRN 5 (i.e. CARDS used as an input address). The use of a standard ROWIN subroutine by all problem programs ensures compatibility of data between all the problem programs. The entry points to the subroutine are ROWIN, TWOIN, ROWOUT, TWOOUT, and HEADER. The first four entry points provide all handling of actual data. ROWIN and TWOIN provide for input into single precision and double precision buffers respectively. ROWOUT and TWOOUT provide output from single precision and double precision buffers respectively. Entry HEADER is used to input the header record of a SOUPAC written data set whenever header record information is desired without also calling for the first row of data.

Each data set handled by ROWIN consists of a header record as the first logical record of the file, and a separate logical record for each row of the data matrix. The header record consists of three four-byte variables: NROW, the number of rows of the data matrix (i.e. remaining number of logical records in the file after the header record); NCOL, the number of variables per row; and LMODE, a logical variable which indicates whether the data matrix is written on secondary storage in single precision (i.e. LMODE is false) or double precision (i.e. LMODE is true). Values for NCOL and LMODE are always correct in the header record since they are both known before the data set can be written. The value for NROW, however, is sometimes not known at the start of a data set. When its value is not known by the problem program, NROW is passed to ROWOUT or TWOOUT with the value zero, and so NROW is written in the header record with the value zero. In such cases, a value for NROW is not known until the entire

matrix has been output. At that time, the appropriate OUT routine is called one extra time, and a correct value for NROW can be saved.

The problem is: where can the value for NROW be saved? For direct access method files (i.e. DISK addresses) NROW can be written into record one of the data set. For sequential access method files, however, this solution is not feasible, so instead the value for NROW is stored in a table in subroutine OUTSET in the I/O monitor (see section II.B). The reason that rewriting the header record for a sequential file is not feasible is that if the file is on magnetic tape, it is not possible to guarantee that the rewritten first record will be written in exactly the same place that the header record was written when the file was created.

ROWIN and TWOIN have seven arguments;

the input address,
the number of rows of the data matrix,
the number of columns of the data matrix,
the current row number,
the buffer into which the data is to be read,
a variable which indicates whether data was transmitted on the current call or whether the end of the data set was found,
a logical variable which indicates the mode which the data is stored on secondary storage.

Arguments one and four are provided by the calling program to ROWIN or TWOIN. The absolute value of the first argument is a DSRN and therefore must be in the range 1 thru 99; in particular, if the input address is a zero, an error message will be printed and the job will be terminated via CALL ERROR. Note that only the bottom order halfword of the input address is used for the DSRN. The fourth argument, the current row number, must always start with the value one when the first row of data of a matrix is desired since ROWIN and TWOIN know to read the header record only when the row index is one. If, however, the first row of data is being read and the header record has already been processed by a call to HEADER, the input address should be the negative of the DSRN to be used so that the subroutine

knows that the header record has already been read. Otherwise, ROWIN and TWOIN will automatically read the header record whenever the first row of data is requested. When the input address is passed negatively, ROWIN and TWOIN set the passed value positive so that it will be correct on subsequent calls for rows of data. Whenever the input address is for card image input, the DATA format card is read by a call to FREAD1 as the "header record" for the card deck (see section IV.D.3).

Variables two, three, six, and seven are initially returned to the calling program by ROWIN and TWOIN. Variables three and seven, NCOL and LMODE respectively, are always known from the header record so that after the header record has been processed, NCOL and LMODE are effectively passed back from the calling program to ROWIN and TWOIN, and consequently only variables two and six are still returned to the calling program from ROWIN and TWOIN.

NROW will always be correct in the header record of a DISK file. Whenever ROWIN or TWOIN reads the header record for a sequential file, OUTSET in the I/O monitor is also called and the value for NROW is returned from OUTSET. If NROW is defined both in the header record and in the table in OUTSET, as is most often the case, the table entry in OUTSET is used as the correct value for NROW even though there probably is no discrepancy between the two values when both are defined. If there is no entry in OUTSET, the value for NROW from the header record is used. NROW should be correct from OUTSET if the data set being read was created during the current job step since NROW should always be entered correctly in the table in OUTSET for sequential files created during the job step. NROW will not be known, however, if NROW is zero in the header record of the file being read, the file was created in a previous job step, and a #OLD control card was not included as a prolog card in the user's SOUPAC program

parameter deck (i.e. there is no entry in OUTSET for NROW). Whenever this situation occurs, a warning message is printed explaining that the number of rows to be input from the particular sequential file is not defined. If this warning message appears and the file being read was created during the current job step, there is a SOUPAC staff programming error in the statistical routine which created the file.

Variable six, LAST, will always be zero if data has been read, and will be set to non-zero if the end of the data set has been found. When the end of the data set is found, one is subtracted from the current row index number, the fourth argument to ROWIN and TWOIN, and the result is returned to the calling program for NROW. For this reason, the fourth argument should always be incremented by one each time a new row is requested from a given input matrix, thus insuring a correct row count after the entire matrix has been read. Only when LAST is non-zero is the value returned for NROW guaranteed to be correct for input from all addresses.

Ths seventh argument to ROWIN and TWOIN, LMODE, is never really needed by the problem program. LMODE is passed as an argument to ROWIN and TWOIN merely as a storage location where the mode of the data on secondary storage can be saved from the header record for use on subsequent calls to ROWIN or TWOIN. Data is always read in as four-byte words. If the data is stored on secondary storage in double precision (i.e. if LMODE is true), 2*NCOL four-byte words are read into the buffer provided by the calling program as the fifth argument to ROWIN or TWOIN. If ROWIN was called, the problem program wants the data in single precision so the high order fullword of each doubleword variable is packed into NCOL consecutive fullwords in the fifth argument buffer, ROW; note that this implies no rounding of data values, just straight truncation of the low order fullword. If TWOIN was called and

LMODE is true, the problem program wants the data in double precision, so no packing is performed. If the data is stored on secondary storage in single precision (i.e. LMODE is false) and if ROWIN was called, the data will be read in precisely as the problem program requires the data. If LMODE is false and TWOIN was called, the data is read into the odd numbered (high order) fullwords of each doubleword of ROW, and the even numbered (low order) fullwords of ROW is set to zero. If the data is being read from CARDS, LMODE is ignored since the data is automatically read into ROW in the correct precision by either READ, for use by ROWIN, or TREAD, for use by TWOIN (see section IV.D.2). The target area for the data, ROW, should be dimensioned within the calling program to 900 fullwords or 450 doublewords (i.e. long enough to handle a double precision row of 450 variables, the currently advertised upper bound on the number of variables guaranteed between problem programs) for both ROWIN and TWOIN since even if ROWIN is being called, the data may be stored on secondary storage in double precision and must be read into primary memory in the mode it is stored on secondary storage before it can be packed into single precision.

Entry HEADER is used if the problem program wants to know header record information before the first row of data is read by ROWIN or TWOIN. HEADER has four arguments;

the input address,
the number of rows of the data matrix,
the number of columns of the data matrix,
a logical variable which indicates the mode which the data is stored on secondary storage.

Argument one is provided by the calling program to HEADER. The bottom order halfword of the argument is used as a DSRN and therefore must be in the range 1 thru 99. If the input address is for cards (i.e. DSRN 5), the "header record" information is taken from the DATA format card or cards in front of the card data deck by FREAD2 (see section IV.D.3).

Arguments two, three, and four -- NROW, NCOL, and LMODE respectively -- are returned to the calling program by HEADER. Arguments three and four will always be correctly known from the header record. If input is from CARDS, the mode value is returned false to the calling program, an arbitrary choice since the mode value will be completely ignored by ROWIN and TWOIN if input is from CARDS. All remarks in the discussion of ROWIN and TWOIN concerning the establishing a value for NROW from the header record and from OUTSET apply for entry HEADER.

ROWOUT and TWOOUT have six and seven arguments respectively, the arguments of ROWOUT being the same as the first six arguments of TWOOUT. The seven arguments to TWOOUT are;

the output address,
the number of rows of the data matrix,
the number of columns of the data matrix,
the current row number,
the buffer from which the data is to be written onto secondary storage,
a variable to indicate whether data is to be written on the current call or whether all data in the matrix has already been written, and the current call is to set a correct value for NROW,
a logical variable to indicate whether the data is to be written onto secondary storage in single or double precision.

All arguments are passed from the calling program to ROWOUT or TWOOUT.

For TWOOUT, argument seven, LMODE, indicates whether the data is to be written in single precision (LMODE is false) or double precision (LMODE is true). For single precision output from TWOOUT, only the odd numbered (high order) fullwords of the data row are written out; the even numbered (low order fullwords are skipped over. For double precision output from TWOOUT, the entire data row is written as 2*NCOL consecutive fullwords, where NCOL is the third argument in the calling sequence to TWOOUT. The reason that no mode argument is in the calling sequence to ROWOUT is that there is no gain in writing out single precision data in double precision. By default, all

data written from buffers supplied by ROWOUT are written in single precision. In the case of TWOOUT, NROW, NCOL, and LMODE are used to create the header record when the first row of data is being output. In the case of ROWOUT, since no mode argument is present in the calling sequence, NROW and NCOL and the implied mode value false are used to create the header record. Since ROWOUT and TWOOUT know to create the header record only if the fourth argument, the row index number, is one, it is essential that the row index start with the value one on the first row of data.

Sometimes a problem program will not know the number of rows in the data matrix when the first row of data is being output. In this case, NROW should initially be passed to ROWOUT or TWOOUT with the value zero, so that NROW will be written into the header record and stored into OUTSET as zero. After all rows of the data matrix have been output, ROWOUT or TWOOUT should be called one extra time, with LAST non-zero, so that a correct count of the number of rows can be calculated. The count is determined by subtracting one from the row index number passed when LAST is non-zero. For this reason, the row index number should always be incremented by one each time ROWOUT or TWOOUT is called for a given output matrix, so that when LAST is set non-zero, the row count will be accurately calculated. The calculated value for NROW is then written into record one for direct access addresses or stored in OUTSET in the I/O monitor for sequential access addresses.

The bottom order halfword of the first argument to ROWOUT and TWOOUT, the output address, is used as a DSRN and therefore must be an integer in the range 1 thru 99, with the exception of DSRN 5. If CARDS is specified as an output address, an error message is printed and the job is terminated via CALL ERROR.

The ROWIN subroutine is written so that the problem program does not typically need to concern itself with what the DSRN being referenced is.

The one exception is that some problem programs need to check if input is from CARDS, otherwise the source or destiny of a data matrix is transparent to the problem program. In particular, the ROWIN subroutine takes care of all problems considering whether the data set being referenced used direct access or sequential I/O.

Similarly, the ROWIN subroutine handles all problems of conversion between primary and secondary memory of single and double precision data items. The accompanying ROWIN USAGE TABLE presents a concise reference on precision conversion between (primary) memory and secondary storage. Values for LMODE under the input section are not listed because the value of LMODE from ROWIN and TWOIN is of no significance to the problem program.

It should be noted that the ROWIN subroutine is so constructed that if one wished to simply copy a data matrix from input address IA to output address IB, the following program segment, however useless it might otherwise be, would do the job correctly with a minimum amount of undue concern by the problem program.

```
      I = 0
1     I = I + 1
      CALL TWOIN  (IA,NROW,NCOL,I,ROW,LAST,LMODE)
      CALL TWOOUT (IB,NROW,NCOL,I,ROW,LAST,LMODE)
      IF (LAST.EQ.0) GO TO 1
```

Notice also that the values for NROW and NCOL returned from the first call to ROWIN or TWOIN, or HEADER if HEADER is used, are appropriate for NROW and NCOL in calling MANAGE. If a matrix being read in by calls to ROWIN or TWOIN is to be stored in an array defined by MANAGE, NCOL will be correct for the number of variables per row and NROW will be correct for the estimated upper bound on the number of rows of the data matrix, arguments two and four respectively in the definition call to MANAGE. If NROW is known to ROWIN or TWOIN, then the correct value for NROW will be passed to MANAGE as the upper bound estimate on the number of rows. If NROW is returned with the value zero

ROWIN USAGE TABLE

INPUT

secondary storage (and card input)	to	memory	entry point
single precision	-->	single precision	ROWIN
double precision	-->	single precision	ROWIN
single precision	-->	double precision	TWOIN
double precision	-->	double precision	TWOIN

OUTPUT

secondary storage	from	memory	entry point	LMODE
single precision	<--	single precision	ROWOUT	none
double precision	<--	single precision	- not supported -	
single precision	<--	double precision	TWOOUT	FALSE
double precision	<--	double precision	TWOOUT	TRUE

Note: The entry point names are not specifically related to the manner in which a matrix is stored on secondary storage. Entry point names express the I/O point of view of the problem program. It is the function of the variable LMODE to control mode conversion between memory and secondary storage.

from ROWIN or TWOIN, the same zero value is appropriate for passing to MANAGE, and thereby informing MANAGE that no upper bound on the number of rows is known (see section IV.F).

One additional function which ROWIN performs is in making an entry into the named common area REWCOM each time the header record for a sequential file is processed. In this manner, REWCOM has a list of all sequential addresses used by ROWIN. At the end of execution of the problem program, subroutine IOREW is called from subroutine NEXT. IOREW then rewinds all those sequential files used by ROWIN.

D.2. READ

Subroutine READ, with additional entry point TREAD, is called by ROWIN whenever the input row is to be from CARDS. READ is called if the variables are to be read into a single precision row; TREAD is called if the variables are to be read into a double precision row. The format, stored in array FMT in ROWIN, is the format which has been scanned off of the DATA format card by subroutine FREAD1 (see section IV.D.3).

Subroutine READ has one additional entry point called RCOPY. RCOPY is called by MAIN for all problem programs which contain ROWIN (see section III.A). When RCOPY is called, entry COPY in INTRFACE is called with two arguments, the buffer CBUF and a variable indicating the length, in bytes, of CBUF. During the running of the problem program, INTRFACE copies all card images, up to the size limit of the buffer, for a given read request from DSRN 5 (i.e. CARDS) into CBUF. This enables READ, upon successful completion of a read statement, to print out the card images which were read during the one read statement, if desired, and also permits sequence checking to be done by subroutine SEQCHK, if desired. RCOPY also calls ERRSET, part of the IBM supplied error monitor. In calling ERRSET, RCOPY indicates to the error monitor that in the case of an error 215 (i.e. invalid characters in an I, F, E, or D input field) the exit routine CHKERR is to be called (see section IV.D.4). Entry RCOPY returns directly to MAIN.

After a read has successfully completed, IPAR(72) contains a count of the number of card images read by the one read statement. IPAR(72) is set by INTRFACE (see section IV.A), and its value is saved by READ in variable LROWL. For all rows which are not the last row, IPAR(72) will necessarily have the same count. After each successful read, LEOF is set to false to indicate that the logical end-of-file has not yet been found for the current data deck.

LCOPY is checked to see if the user included a #COPY card. If LCOPY is true, the contents of CBUF, containing the card images saved by INTRFACE are printed out.

If sequence checking was requested, indicated by a non-zero value of IPAR(76), READ calls subroutine SEQCHK. Note that IPAR(76) is equivalenced to SEQOPT within READ; values for SEQOPT are set by FREAD1. If an error is found during sequence checking, READ reads an appropriate number of cards to "correct" for missing or extra cards. In case of either an invalid input character or a sequence error, the observation is discarded, the variable IOERR is incremented by one, and READ proceeds to try to read the next observation. READ proceeds in this manner until it gets a correct observation at which time it returns to ROWIN.

The end-of-file return, END=1, of the read statements cause control to go to statement number 1 (one) whenever an actual end-of-file condition occurs or whenever INTRFACE reads an END# card and simulates the end-of-file return (see section IV.A). At statement 1, variable LEOF is checked for a value of true, indicating that an end-of-file return was executed on the last call to subroutine READ. If a double end-of-file condition exists, an error message is printed and the job is terminated via CALL ERROR. If LEOF is false, however, IPAR(72) is checked for -1, indicating that one card was read and that the one card was an END# card. If IPAR(72) equals -1, LEOF is set to true to indicate that a logical end-of-file has been found, and LROWL is reset to zero in case another card input deck is to be read by the current problem program. At this point, the count of I/O errors (i.e. the variable IOERR) is checked for being equal to zero. If IOERR is zero, READ returns to ROWIN; if IOERR is non-zero, the job is terminated by a CALL ERROR. If IPAR(72) does not equal -1, control goes to statement number 30.

At statement 30, IPAR(72) is checked for zero. If IPAR(72) is zero, this

implies that the current deck was terminated by a /* card and a warning message is printed to that effect. READ then proceeds to return to ROWIN normally. If the end-of-file return was taken and more than one card was read, or if one card was read and that card was not an END# card, either an END# card or an actual end-of-file (i.e. a /* card) was found in the middle of what was meant to be a row of data. In this case, the contents of LROWL, indicating the number of cards read by the last successful input of a full row of data from the current data deck, and a count of the number of data cards read on the current abortive read are printed out in an error message. Note that the END# card, if it was present is not included in the data card count. If LROWL is zero, the end-of-file was found while trying to input the first row of data in the current deck. Finally, the job is terminated by a CALL ERROR.

D.3. FREAD1

Subroutine FREAD1, with additional entry point FREAD2, is called by ROWIN to process the DATA format card at the front of each user card data deck (i.e. whenever CARDS is used as an input address). The difference between FREAD1 and FREAD2 is that FREAD2 requires that some value for the number of logical data rows of the input deck (i.e. NROW) must appear within the first set of parens (i.e. there is a comma within the first set of parens). Upon entry to the subroutine at FREAD1, register 3 is set off; entry at FREAD2 causes register 3 to be set on. Both entry points then go to location BALRORD. Register 3 is then saved in NROWFLAG.

Before proceeding with any card image scanning, DECKNO is increased by one to indicate that a new data deck is being processed. Notice that the location DECKNO being increased, however, is the DECKNO in the SVT not the DECKNO in this subroutine. DECKNO is not addressed directly, but rather is referenced by an address constant in DECKADDR as a pointer to DECKNO. The MAIN macro, for all problem programs which have ROWIN, stuffs the address of the DECKNO in the SVT into DECKADDR (see section IV.A). Note that DECKADDR is an external reference in FREAD1. The reason that there is a DECKNO also in the subroutine, even though it is never normally used, is that its presence makes easier use of FREAD1 outside the context of the entire SOUPAC system. The DECKNO in the SVT is used in printing out error diagnostics by the loader during error termination of a SOUPAC job (see section II.A and III.B).

After increasing DECKNO by one, FREAD1 now calls the internal routine NEWCARD to read in the first data format card. NEWCARD reads a card, performs an end-of-file return to EOFERR if an end-of-file condition occurs, prints out the card image just read and then proceeds to remove all blanks from the card image. If control goes to EOFERR, a message is printed out and

the program is terminated by a CALL ERROR.

Upon the first return from NEWCARD, the first four characters of the image, after the blanks have been taken out, are checked for the characters DATA. If the first four non-blank characters are not DATA, control goes to location ERROR1, an error message is printed out, and the job is terminated via CALL ERROR. Otherwise, scanning continues by calling the internal routine CHECK via the internal macro LOOK to find the first left paren.

The LOOK macro has two arguments. The first argument is the character to be scanned for. This character is moved into location SEARCH+1. The second argument is either ERROR2 or ERROR3 and is the location control is to go to if the end of a card image is reached before the desired character is found. ERROR2 is used if the character scanned for is a beginning left paren. ERROR3 is used if the character being scanned for is a right paren. The difference between ERROR2 and ERROR3 is that if a left paren is being looked for and has not been found by the end of a card, that card contains no remaining useful information and can be ignored. Note that register 3 is used as a base register for the buffer CARD, however register 3 is continually incremented as characters are scanned. Hence, the current character is always referenced as CARD even though the character is somewhere in the middle of the buffer. ERROR2 causes register 3 to be reset to point to the beginning of the buffer by loading an address constant from location ADDRESS. This has the effect of causing the new card image to be read into memory beginning at the actual location CARD. In the case of ERROR3, register 3 is not reset; a new card read in will be read into the next available location after the current card image. ERROR3 is branched to only in the one case of looking for the right paren after the first left paren.

After the first left paren is found, the next character scanned for via the LOOK macro is the next right paren. Location LEGAL# is set to

indicate that the internal routine CHECK is to allow between the left and right parens only characters 0 thru 9. The only exception to this rule is that a single comma is permitted between the left and right parens. Location COMMA# is used to make sure that at most one comma appears between the two parens. If any character other than the numbers 0 thru 9 or a single comma is used, control goes to location ERROR6, an error message is printed out, and the job is terminated via a CALL ERROR.

After the right paren is found, values for the number of logical data rows (i.e. NROW) and the number of variables (i.e. NCOL) are determined with the aid of the internal routine CONVERT. If no comma is found between the two parens, and if FREAD2 was called as the entry (i.e. NROWFLAG+3 is on), control goes to location ERRORO, an error message is printed out, and the job is terminated via a CALL ERROR.

At this point, FREAD goes looking for the next left paren. This parenthesis will be the beginning of either a sequence check specification or the data format. When the left paren is found, the current card image is left justified within the buffer CARD by a call on LEFTJUST. Next the variable SEQOPT, equivalenced to IPAR(76), is set to zero indicating that no sequence options have yet been found for the current data deck. As options are found, SEQOPT will be set to indicate which options were specified. If no sequence checking is specified by the user, SEQOPT will remain zero.

FREAD1 determines if sequence checking is to be performed by comparing the first three characters after the left parenthesis with the first three characters of each of the four sequence checking options. If no match is found in the four compares, the left parenthesis is assumed to be the beginning of a format, and control goes to location ZAGREB.

If a match is found, sequence checking has been specified, and the various possible options are checked for, and appropriate values to SEQOPT are assigned. Options are separated by the semi-colon, and the end of the sequence checking specification is indicated by a right parenthesis. The parameters as found on the data format card are transformed into a form which can be directly used by subroutine SEQCHK (see section IV.D.5). For example, the beginning column specification for IDCOL= and SEQCOL= is converted from character to integer and decreased by one. The reason for decreasing the column number by one is that the first byte of a card image is addressed in SEQCHK by its number of bytes past the beginning of the card buffer, which is zero (i.e. the first byte begins at zero bytes past the beginning of the card buffer). Field lengths in IDCOL= and SEQCOL= are similarly converted to integer and are decreased by one to be appropriate lengths which can be directly used in CLC and MVC instructions. For the IDOPT= option, a BRANCH instruction is set up with the correct condition code corresponding to the option chosen (i.e. SI corresponds to a branch-not-high, MI corresponds to a branch-on-low, SD corresponds to a branch-not-low, MD corresponds to a branch-on-high, and U corresponds to a branch-on-equal). Additional error checking and parameter setup is performed, and all resulting parameters are left in the table SEQPARMS. SEQPARMS is an entry in FREAD1 so that subroutine SEQCHK can reference SEQPARMS as an external reference to determine what to do. After all sequence checking and building of SEQOPT and SEQPARMS has been performed, FREAD1 goes looking for the left parenthesis which begins the format. As soon as the left parenthesis is found, the remainder of the card image is left justified by a call to LEFTJUST.

Actual scan of the format begins at location ZAGREB. Register 5 is set to one to indicate that one left parenthesis was found. As additional left parentheses are found, one is added to register 5; as right parentheses are found, one is subtracted from register 5. Scanning continues until register

5 goes to zero, indicating that a matching number of left and right parentheses have been found. A maximum of 592 characters (an arbitrary choice) are allowed for the format string. This size is dependent upon the dimension of the array FMT within subroutine ROWIN. If register 5 does not go to zero before 592 characters are scanned, control goes to location ERROR4, an error message is printed out, and the program is terminated via a CALL ERROR. Note that if the format string continues over more than one input card image, each card image is shifted into the next available location in FMT, pointed to by register 2. Register 3 is therefore reset from location ADDRESS and each new card read during the format scan is read into the beginning of the array CARD. Register 7 is used to tick off characters as they are scanned, and it is when register 7 goes to zero that control goes to location ERROR4.

Examples of legal data format cards:

```
DATA (20) (10F8.0)
DATA (20,20) (10F8.0)
DATA FOR GROUP ONE WITH (20) VARIABLES UNDER FORMAT (10F8.0)
D A T A (,20) (10F8.0/10F8.0)
DATA (20) ((10F8.0))
DATA (20) (IDCOL=1,4;SEQCOL=5) (5X,10F6.0)
DATA (20) (IDCOL=1,5;IDOPT=SI;SEQCOL=6,1;SEQSET=A,B,C) (10X,6F10.0)
```

Examples of illegal data format cards:

```
DAT (20) (10F8.0)
DATA (20,20,20) (10F8.0)
DATA (10F8.0)
DATA (TWENTY) (10F8.0)
DATA (20) ((10F8.0)
DATA (20) ( ) (10F8.0)
```

D.4. CHKERR

CHKERR is used in conjunction with READ to handle format conversion errors. When entry RCOPY is called from MAIN, RCOPY calls ERRSET, part of the IBM supplied error monitor. RCOPY supplies CHKERR to ERRSET as the error exit routine to be called in the event of an error 215. Error 215 is sensed by IBCOM# and indicates an illegal character in an I, F, E, or D input field.

Whenever an illegal character is found by IBCOM# in an I, F, E, or D input field, CHKERR is called by the error monitor. CHKERR first indicates a return code zero, indicating to the IBM error monitor to take the "standard corrective action." CHKERR then saves in BADIO the number of the card within the current data deck of the card in error. This information is then used by READ in printing a diagnostic message (see section IV.D.2). CHKERR returns to the IBM error monitor.

D.5. SEQCHK

Subroutine SEQCHK is called from READ to perform sequence checking of input data whenever this feature has been indicated by the user (see sections IV.D.2 and IV.D.3). SEQCHK determines which of the four options were specified by values of SEQOPT. Parameters to SEQCHK come from the table SEQPARMS, an entry in FREAD1.

First, SEQCHK determines the number of cards in CBUF, where CBUF is referenced within this subroutine by its named common name ASDF. This is done by taking the minimum value of LROW and MAXREC, LROW indicating the number of cards read on the current read request, MAXREC indicating the maximum number of cards CBUF can hold.

Next a test is made to see if IDCOL= was specified. If IDCOL= was not specified, id checking is skipped entirely. If IDCOL= was specified, the id field is checked for consistency in all the card images in CBUF. Then a test is made to see if IDOPT= was specified. If it wasn't specified, the option is skipped. If the current row is the first row of data, there is no previous id with which to compare the current one so no comparison is made. If, however, the option is requested and the current row is not the first one, the correct BRANCH instruction is loaded from SEQPARMS and stored at BRANCH and a test is made of the current id with the id from the previous row. If the test succeeds, the current id replaces the previous id and execution continues.

Explicit checking of sequence numbers within a logical row of data is performed next. If SEQCOL= was not specified, this section is entirely skipped and the subroutine returns to READ. A test is made to see if SEQOPT= was specified. If so, the sequence values are taken from SEQPARMS and are the character values as scanned off the data format card by FREAD1. If SEQOPT= was not specified, the sequence values are taken from the table NUMTAB which merely

has the integers 1 thru 20 in character mode. In either case, the register WK1 is used to point to the correct table. Notice that sequence checking is performed entirely in character mode. Next the cards in CBUF are indexed thru, front to back, checking that the sequence values in the data cards correspond to the tabled values. If all values correspond, SEQCHK returns to READ.

SEQCHK can return to READ in one of two error conditions. The first error condition indicates that an id error occurred, the second indicates that a sequence error occurred. Note that if an id error occurrs, the program never performs the sequence check portion. In the case of either error, SEQCHK returns to READ two integer values. The first integer indicates which card-image in the current logical row was determined to be in error. The second integer indicates where such a card image should probably appear in a logical row. READ uses this information to determine whether or not it needs to skip some card images in an attempt to recover from the error, and if so how many should be read (see section IV.D.2).

D.6. IOREW

Subroutine IOREW is called from subroutine NEXT and rewinds those sequential files used, indicated by a value "time" in the logical array IOADDR, which were used during execution of the problem program.

D.7. Reading a SOUPAC data set outside SOUPAC

The question often arises, "How can I read a SOUPAC created tape using my own FORTRAN program?" The procedure is simple, assuming a basic knowledge of IBM FORTRAN input/output conventions.

1. Most SOUPAC input/output is done using calls to TWOIN, TWOOUT, ROWIN, and ROWOUT.

2. TWOIN (etc.) uses unformatted I/O.

3. The first record of each file is the "header record," consisting of NROW, NCOL, and LMODE.

4. All subsequent records begin with three 4-byte words of essentially dummy information (NROW, NCOL and LMODE again).

5. After the three full words as described in 4 above, data records (all records after the first) consist of NCOL variables, each item either 4-bytes or 8-bytes. The record consists of 4-byte items if LMODE is 0, and contain 8-byte items otherwise.

6. The exceptions to the above rules are only the special purpose I/O functions found in TRANSFORMATIONS and MATRIX.

In order to read a file created by TWOOUT, determine whether the data has been written in single or double precision. In general, most files are written in double precision, except TRANSFORMATIONS always uses single precision. A SOUPAC consultant can help you with this.

Proceed to read the first record as follows:

```
READ (11) NROW, NCOL, LMODE
```

Notice the use of no format. Subsequently, you may read data records as

```
READ (11,END=100)N1,N2,N3,(ROW(J),J=1,NCOL)
```

After the entire file has been read, control will go to statement 100. ROW should be dimensioned REAL*4 or REAL*8 depending upon the precision of the data on the tape.

It is possible to read the data without knowing the precision of the individual data items (by determining the precision from the header record value of LMODE), but this need be attempted only by experienced programmers. (See section IV.D.1 on ROWIN.)

No header record is generated by the OUTPUT instruction in MATRIX, so that reading data from that file would be performed using ordinary IBM FORTRAN I/O conventions. Also, the TRANSFORMATIONS "special-purpose" input/output features will not have header records, and likewise follow standard IBM FORTRAN conventions. In all cases remember that one row of data constitutes one logical record of the data file.

When preparing to read a SOUPAC file, you will also need to prepare appropriate JCL cards to point to the correct data set. In the SOUPAC procedure, you will want to override the DD statement for the file to be later read providing a data set name, and other pertinent information.

In the FORTRAN go step, you will then need a DD card pointing to the data set to be read. The DD name in the FORTRAN go step will need to correspond to the Data Set Reference Number used in the FORTRAN program. For example, if the file is to be read using the READ statements as indicated above, the DD statement which points to the desired data set should have name FT11F001.

E. BCNVT

BCNVT represents the one departure by SOUPAC from the IBM 360 FORTRAN format conversion standards. On formatted input, BCNVT causes blank fields to be read in as minus zero; on formatted output, BCNVT causes minus zero to be printed out as minus zero. Normal IBM 360 FORTRAN (Release 17) reads in a blank field as true zero and prints out minus zero as true zero. The single word hexadecimal representation of minus zero is 80000000.

Whenever a format conversion is required, the routine desiring the format conversion loads the address of the conversion routine from a table at ADCON#. In particular, the first four words of ADCON# contain the address of the conversion routines for F format input, F format output, E format input, and E format output respectively.

To get the routines in BCNVT called before the regular conversion routines are called, the MAIN macro saves the first four words of ADCON# in the table CTABO in BCNVT and then shifts into the first four words of ADCON# a table which points to the appropriate entries in BCNVT (see section III.A). When an E or F format conversion is then requested, control will go to the routine in BCNVT, and BCNVT then will call the actual IBM conversion routine if necessary.

Entry points MZFIN and MZEIN are used to check formatted record input data fields which are being converted under F and E formats respectively. If the data field is entirely blank, BCNVT returns a value of minus zero to its caller, IBCOM#. If any non-blank characters are found in the field, BCNVT immediately calls the actual IBM conversion routine by using the appropriate address placed in CTABO by MAIN. The actual conversion routine then performs the conversion it would normally do and then returns directly to IBCOM#.

Entry points MZFOUT and MZEOUT are used to perform formatted record output data conversion. When BCNVT is called thru these entry points, the variable to be converted is checked for minus zero. If the variable is not minus zero, the regular conversion routine is called so that it will return directly to IBCOM#. If the variable to be converted is minus zero, the regular conversion routine is called, however it is called so that it will return to BCNVT. BCNVT then checks to see if the field in the output buffer begins with an asterisk. If this is the case, BCNVT returns directly to IBCOM#. If the first character is not an asterisk, BCNVT then checks to see if the first character is a blank. If it is not blank, there is no room in the field to put in a leading minus sign. In the case of no leading blank, control goes to location ASTINSRT and the entire field is then filled with asterisks. Filling a field with asterisks is the IBM FORTRAN convention for indicating that a variable could not be represented in the format field specified.

If the first character is a blank, BCNVT proceeds left to right thru the remainder of the field and finds the first non-blank character. This character must necessarily be the character 0. A minus sign is put in front of the 0 and BCNVT returns to IBCOM#.

F. MANAGE

Subroutine MANAGE is used to reference data matrices by SOUPAC problem programs which perform run time storage allocation (see section III.C). Entry point MANAGE is not called to handle data directly but is called before any data is handled to define the size and other characteristics of a data matrix. Entry MANAGE is called once for each separate data matrix that the subroutine is to handle. A maximum of ten matrices may be defined by calls to MANAGE.

After all matrices have been defined by calls to MANAGE, rows of data, the basic work unit in the management scheme, are referenced by the problem program thru the remaining entry points of MANAGE; hereafter referred to as the X$, A$, B$, and C$ entry points. There are five A$ entry points; AIN, AOUT, A$RLSE, A$INIT, and A$PUNT, and there are correspondingly five entry points for B$, C$, and X$. All A$, B$, and C$ entry points have two arguments; the number of the row to be referenced, and an index pointer to where that row is located in primary memory. The five corresponding X$ entry points have an additional third argument. The third argument indicates by number which matrix, of all those defined by calls to MANAGE, the current row request is for. In particular, if the third argument is one, the X$ routines will return the same index pointer as would be returned by calling the corresponding A$ entry points. Similarly, matrix number two corresponds to the B$ entry points; matrix number three corresponds to the C$ entry points. Clearly the X$ entry points can do anything the A$, B$, and C$ entry points can do; the others are included for the coding convenience of the SOUPAC staff programmer.

The basic premise of MANAGE is that as many complete rows as possible of a data matrix are kept in primary memory; remaining rows are written onto secondary memory and are rolled in and out as needed. Rows stored in

secondary memory are referenced thru DSRN 99 using direct access method I/O. Calls to entry MANAGE define the data matrix to be handled and provide the information necessary to determine how many rows of a data matrix will fit into primary memory. The first call to MANAGE defines the first matrix, thereafter referenced thru the A$ entry points; the second call to MANAGE, if one is made, defines the second matrix, thereafter referenced thru the B$ entry points; an additional call would define a third matrix and would be referenced thru the C$ entry points. Although a maximum of ten matrices may be defined for MANAGE, only the first three matrices have their own permanently assigned entry points. The remaining seven matrices must be referenced thru the X$ entry points with the matrix number provided as the third argument on all X$ calls.

There are six arguments in the calling sequence to MANAGE. The first argument is the beginning address of the block of primary memory that MANAGE is to consider for use by the current data matrix. The second argument is the number of variables per row of the current data matrix. The third argument is the number of words in the primary memory block. This number will be the number of four byte words if the matrix is a single precision matrix or will be the number of eight byte words if the matrix is a double precision matrix. The fourth argument is an upper bound estimate on the number of rows of the data matrix. If no upper bound is known, this argument should be zero. The fifth argument is the address of an array which can be used by MANAGE to contain information about the roll buffers. The sixth and last argument is a one if the matrix is a single precision matrix or a two if the matrix is a double precision matrix.

The array specified as the fifth argument is necessary for the proper execution of the secondary storage management scheme. This array is used to keep roll memory information for a matrix. The first word of the array

contains an integer which indicates how many roll buffers must be available in primary memory to handle rows stored in secondary memory. For each roll buffer for a data matrix, there must also be an additional full word in the array which appears as the fifth argument to MANAGE. The integer in the first word of the array will therefore always be one less than the dimension of the array. A matrix that requires two roll buffers would have a three word array as the fifth argument to MANAGE and the first word of the array would contain the integer two. Each full word of this roll buffer information array, other than the first word, is used as two halfwords. The botton order halfword is used to keep the record number on DSRN 99 of the row which is currently in the corresponding roll buffer. The top order halfword is used to keep the responsibility count for the row of data in the corresponding roll buffer.

The question of the number of roll buffers needed and the problem of keeping a responsibility count for a record in a roll buffer are closely related. For example, if a problem program needs at any one time the I^{th}, J^{th}, and the K^{th} rows of a matrix, three roll buffers will need to be specified since if all three rows happen to be in secondary memory, it will be necessary to have a separate roll buffer for each row. If the values for I, J, and K, however, happen to be the same during the execution of a program, it is not desirable to have three copies of the particular row, one in each work buffer. Instead a responsibility count is set to indicate that the particular row is being used under the control of three separate indices. The number of roll buffers to be specified for a matrix will be the same as the maximum responsibility count which any one row could have at any one time.

The five entry points of the X$, A$, B$, and C$ entries are used to do the actual data referencing. To get a pointer to the I^{th} row of the first

matrix, the problem program would code

CALL A$IN (I,IR)

In IR is returned a pointer to the I^{th} row such that the J^{th} element of the I^{th} row would be referenced as A (IR + J). Note that consecutive elements of the same row are stored in consecutive locations. If the I^{th} row is not in primary memory at the time it was requested, a free roll buffer is found and the row is read into the free buffer. A free roll buffer is known to be available if the responsibility count for the buffer is zero.

After the problem program is done with the I^{th} row it must indicate this fact to MANAGE by executing

CALL A$OUT (I,IR) or A$RLSE (I,IR)

If the I^{th} row is normally kept in primary memory, A$OUT will simply execute a return to the problem program. If the I^{th} row is normally kept in secondary storage, one is subtracted from the responsibility count and if the resulting count is zero, the row is written back out to secondary memory. If the resulting count is not zero, the row is still being used and is not written onto secondary memory.

A$RLSE is used in place of A$OUT in the case when it is known that the contents of the row have not been changed since the row was requested. When A$RLSE is called, the row number is checked to see if the row is normally kept in primary memory. If the row is normally kept in primary memory, A$RLSE simply executes a return to the problem program. If the row is normally kept in secondary memory, the responsibility count for the row is reduced by one before returning to the problem program. The difference between A$OUT and A$RLSE is that for rows normally stored in secondary memory, A$OUT writes the row back onto secondary memory when the responsibility count for the row goes to zero while A$RLSE does not.

Note that a problem can arise when using A$OUT and A$RLSE injudiciously. Suppose a roll buffer has a responsibility count of two. If A$OUT is called the count will be reduced to one, but the row will not be written out yet. If A$RLSE is next called, the count will go to zero and the row will not be written out even though it was intended by the previous call to A$OUT that the row be written out when the responsibility count went to zero. This problem is easily avoided by restricting the use of A$RLSE to those instances where it is certain that no changes could have possibly been made to the current row being released.

A$PUNT is similar to A$IN in that it returns a pointer to the current row and increases the responsibility count for a row in a roll buffer if the row is normally stored on secondary storage. A$PUNT is different from A$IN, however, in that A$PUNT will not read into memory the old copy of the current row from secondary storage. A$PUNT saves executing the read request whenever we are not concerned with the previous contents of a row.

The fifth A$ entry, A$INIT, is used to initialize a row to zero without any other complications. If the row is in primary memory, it is set to zero. If the row is stored in secondary memory, a roll buffer is set to zero, and the zeroed row is written onto secondary memory.

One additional entry point is provided to MANAGE. This entry point, MSET provides the ability to redefine data matrix usage for the allocation system. By calling MSET, subsequent calls to MANAGE redefine the data matrices beginning with matrix one. MSET has no parameters.

When MANAGE is called, a counter kept in IDSAVE is increased by one. IDSAVE indicates how many matrices have been defined by calls to MANAGE. Note that it is IDSAVE that is reset to zero whenever MSET is called. Registers 5 thru 10 are then loaded with the addresses of the six arguments to MANAGE. Register 5, containing the address of the block of primary memory

available to the current matrix is stored in ADDRTAB. ADDRTAB and all other tables used for storing matrix defining information has ten entries available for accomodating ten matrices. The number of variables pointed to by register 6 is stored in the table NCOL. The mode, one for single and two for double, which is pointed to by register 10 is multiplied by four to get the number of bytes per variable and the result is stored in the table MODE. Note that NCOL and MODE are halfword tables which respectively occupy the high and low order halfwords of the same ten consecutive fullwords. Next the number of words of the primary memory block is divided by the number of variables per data row to get the maximum number of rows which will fit into primary memory.

A test for zero is made of the estimated upper bound of the number of rows, pointed to by register 8, to see if an estimate was specified. If the estimate has not been specified, control goes to location SUB and roll buffers are saved. If an upper bound estimate has been specified and if it is greater than the number of rows of primary memory available, roll buffers are saved. If an estimate is made and if the estimate is less than the number of rows of available primary memory, no roll buffers need to be saved. The number of rows of the matrix which can be stored in primary memory, not counting roll buffers, is stored in the table IROW. All information is now known by MANAGE so that MANAGE can store or locate any row of the define matrix.

The X$, A$, B$, and C$ entry points each first establish the correct matrix number for the matrix to be referenced and then proceed to execute the code for the appropriate option. There are two macros, DENTO and DENT1, which set up the five entry points for each of the X$, A$, B$, and C$ entry points. Both macros have one argument indicating whether the entry points to be generated are for X$, A$, B$, or C$. The macro DENTO with argument A$

creates the entry points A$IN and A$PUNT; the macro DENT1 with argument A$ creates the entry points A$RLSE, A$INIT, and A$OUT. For the remainder of the discussion on MANAGE, we will use the A$ entry points as examples; the other entry points generalize from the A$ entry discussion.

Upon entry at A$IN or A$PUNT, location SWITCH is set to X'00' or X'FF' respectively indicating which entry was used. The matrix number is set in register ID and control goes to location ALLIN. The row number being requested (i.e. the first argument in the calling sequence to either A$IN or A$PUNT) is compared against the table entry in IROW corresponding to the first matrix. If the requested row is not greater than the IROW table entry, the row is kept in primary memory and a pointer can be returned which points directly to the requested row. The formula for calculating the pointer is:

$$answer = (i - 1 - nbuf) * ncol$$

where i is the row requested.
nbuf is the number of roll buffers saved, if any.
ncol is the number of variables per row.
and answer is the resulting index pointer.

Note that

1) The roll buffers are always accounted for, and by implication the roll buffers are therefore contained in the first nbuf rows of primary memory.

2) The above pointer calculation is independent of the mode of the matrix. What is being calculated is a FORTRAN useable index, not an actual address.

If the requested row is greater than the entry in IROW, the requested row is kept on secondary memory and therefore control goes to location FETCHROW. At FETCHROW, the internal routine SETREC is called to calculate the record number on DSRN 99 for the currently requested row. The formula for the record number calculated by SETREC is

$$nrec = (i - 1 - irow) * nmat + id$$

where i is the row requested

irow is the number of rows of the matrix which are always kept in primary memory

nmat is the number of matrices which have been defined by calls to MANAGE

id is the matrix number of the current matrix

and nrec is the resulting record number.

The resulting value nrec is left in register I.

After the record number for the current row has been calculated, the roll buffer information array, passed as the fifth argument to MANAGE when the matrix is defined, is checked to see if the requested record is already in a roll buffer. If the record is in a roll buffer, control goes to location SETANS. If the record is not in a roll buffer, the internal routine FINDFREE is called and a free roll buffer (i.e. a roll buffer with a responsibility count of zero) is searched for. If no free roll buffer is found, either not enough roll buffers were specified in the fifth argument to MANAGE or the responsibility count was not decreased by a call to A$OUT or A$RLSE. In either case, no free roll buffer is a SOUPAC staff programming error.

After a free roll buffer has been found, location SWITCH is checked to see if entry was via A$IN or A$PUNT. If entry was via A$PUNT, control goes to location SETANS; if entry was via A$IN, the desired record is read into memory and then control falls thru to location SETANS. At SETANS, the proper index pointer is calculated which will allow referencing the row in the roll buffer, and the responsibility count for the roll buffer is increased by one.

Upon entry at A$RLSE, A$INIT, or A$OUT, location SWITCH is set to X'00', X'F0', or X'FF' respectively indicating which entry was used. The matrix number is set in register ID and control goes to location ALLOUT. After ALLOUT, the row number requested is loaded into register I and a check is made of location SWITCH to see if entry was via A$INIT. If entry was via

A$INIT, control goes to ALLINIT, otherwise execution continues. At ALLINIT the row value in register I is checked against the IROW table entry to see if the row is kept in primary or secondary memory. If the row is kept in primary memory, the beginning address of the row is calculated by a call to the internal routine BUFADDR, the row is zeroed by a call to the internal routine ZEROBUF, and control goes to OUTQUIT which returns to the problem program. If the row is kept in secondary memory, control goes to RECINIT. RECINIT calls the internal routine SETREC to calculate the record number for the current row, calls the internal routine FINDFREE to find a free roll buffer, calls the internal routine BUFADDR to calculate the beginning address of the roll buffer, calls the internal routine ZEROBUF to zero out the buffer, calls the internal routine WRITEREC to write out the record, and branches to location OUTQUIT which returns to the problem program. Note that in the internal routine ZEROBUF the instruction to do the storing zero is an STE instruction if the matrix is in single precision and is an STD instruction if the matrix is in double precision.

If entry was not via A$INIT but was either A$RLSE or A$OUT, the row being requested is compared against the appropriate entry in IROW to see if the row is kept in primary memory or secondary memory. If the row is kept in primary memory, control goes to location OUTQUIT which returns directly to the caller. A row of data which is always stored in primary memory is treated the same by A$OUT and A$RLSE.

If the requested row is kept in secondary memory (i.e. if the value in register I is not greater than the value in IROW), a call is made to the internal routine SETREC and the record number corresponding to the current row is computed by the formula already described. A check is then made to locate the roll buffer containing the computed record. If no buffer contains the record, a SOUPAC staff programming error has been made. Once the roll

buffer containing the record has been located, the responsibility count for the record is decreased by one. If the responsibility count after subtraction is still positive, the row is still being referenced within the problem program and so control goes to location OUTQUIT; OUTQUIT returns directly to the problem program.

If the responsibility count has gone to zero, location SWITCH is checked to see if entry was via A$RLSE. If entry was via A$RLSE, control goes to OUTQUIT which returns to the problem program. If the responsibility count is zero and entry was via A$OUT, the record is written out onto DSRN 99. Two internal routines are called to write out the record. First, BUFADDR is called to calculate the beginning address of the roll buffer to be written out. Second, WRITEREC is called to actually write out the row beginning at the address calculated by WRITEREC. After the row has been written out, control goes to location OUTQUIT which returns to the problem program.

G. DREAD and DMIN

Subroutines DREAD and DMIN are used to input an entire double precision data matrix into a problem program from an external source. DREAD is used if the array to be stored into was dimensioned within the problem program; DMIN is used if the array to be stored into was defined by calling MANAGE (see sections III.C and IV.F).

DREAD has six arguments;

the input address,
the number of rows of the data matrix,
the number of columns of the data matrix,
the maximum number of rows which the memory array can store,
the array in memory to be used to store the data matrix,
and a work buffer for reading in each row of data before its being copied into the memory array.

Besides the matrix returned in the fifth argument array, arguments two and three are returned to the calling program by DREAD; the remaining arguments are all supplied to DREAD by the calling program.

DMIN also has six arguments all of which, except for argument number four, are the same as the arguments to DREAD. The fourth argument to DREAD, the maximum number of rows allowed, is not needed by DMIN since rows which do not fit in primary memory are rolled onto secondary memory as needed with no practical upper bound on the number of rows. Instead, DMIN as a fourth argument needs to know the matrix index number so that in calling X$PUNT and X$OUT the correct matrix is referenced.

Both DREAD and DMIN call TWOIN to input each row of data into the work buffer supplied by the calling program. DREAD copies the data row from the work buffer directly into the memory array. DMIN, because it does not know where in memory a row is stored, calls X$PUNT to get a pointer to the desired array row, copies the data from the work buffer into the array row, and then calls X$OUT to get the array row written onto roll memory if necessary (see section IV.F). TWOIN is called repetitively until LAST is

not equal to zero. In the case of DREAD, if the limit on the maximum number of rows allowed is reached before LAST is non-zero, an error message is printed out and the job is terminated via CALL ERROR. Notice that the input address passed to TWOIN equals only the bottom halfword of the input address passed to DREAD or DMIN. This prevents any stray flags being passed to TWOIN in the upper halfword of the input address by mistake.

H. DWRITE and DMOUT

Subroutines DWRITE and DMOUT are used to output an entire data matrix from a double precision array to an external address via calls to TWOOUT (see section IV.D.1). Both subroutines are also used to print out a matrix in either E or F format and to punch out a data matrix. Matrices are printed, with column and row numbers, in blocks of nine columns each with either the format (' ',I3,9E13.5) or (' ',I3,9F13.5). The I3 field is used to print out the current row number and the remaining nine format fields are used for the data. Matrices are punched out a row at a time under the format (2I5,5D14.7). The 2I5 field is used to identify the current row number and the card number for the current card within the current row; the remaining five format fields are used for the data. Note that since both punching the data and outputting the data via calls to TWOOUT are both done a row at a time, these functions are performed first and then printing in nine column blocks is performed in a second half of the two subroutines. DWRITE is used if the array to be written from was dimensioned within the problem program; DMOUT is used if the array to be written from was defined by calling MANAGE (see sections III.C and IV.F).

DWRITE has eight arguments;

the output address,
the number of rows of the data matrix,
the number of columns of the data matrix,
the maximum number of possible rows of the matrix (i.e. the row dimension of the array that the data matrix is stored in),
the array that the data matrix is stored in,
a work buffer,
a logical variable indicator which indicates whether the matrix is a lower triangular matrix (true if it is lower triangular),
and a logical variable which indicates the mode that the matrix is to be written out on secondary storage (false for conversion from double in memory to single on secondary storage; true for no conversion, i.e. double in memory to double on secondary storage).

All parameters are passed from the calling program to DWRITE.

DMOUT also has eight arguments all of which, except the fourth argument, are the same as the arguments to DWRITE. The fourth argument to DWRITE, the maximum number of rows of the data matrix, is not needed by DMOUT. Instead, DMOUT requires as the fourth argument the matrix index number appropriate for use in calling X$IN and X$RLSE (see section IV.F).

The output address passed as the first argument to both subroutines is a fullword which is treated as two separate halfwords. The bottom order halfword contains the DSRN to be used in calling TWOOUT. If the DSRN is zero, TWOOUT is not called. The top order halfword is used as a set of flags controlling printing and punching of the data matrix. Only the bottom three bits of this flag halfword are referenced by DWRITE or DMOUT. The possible bit combinations of the top order halfword, in hexadecimal representation, and their corresponding meanings are:

0000	No printing or punching
0001	Print in E format
0002	Punch
0003	Print in E format and punch
0005	Print in F format
0007	Print in F format and punch

Note that the F format flag cannot be on if the print flag is not also on. If the top halfword of the first argument to DWRITE or DMOUT is zero, no printing or punching occurs. Getting E or F format is implemented in the two subroutines by adding the format flag, 0 or 1, to the character representation of E and putting the result in the format string $FORM; this is done by the FORTRAN statement

```
$FORM(3) = $E + (MOD(FLAG,8))/4
```

where (MOD(FLAG,8))/4 has the value either 0 or 1.

Whenever DWRITE or DMOUT punches a data deck, two additional cards besides the actual data are punched. A DATA format card with the number

of rows and columns of the data matrix and with the format (10X,5D14.7) is punched out as the first card of the deck. The deck is terminated by punching an END# card after the data. Since the number of rows and columns of the data matrix are passed as the second and third arguments to DWRITE and DMOUT, the DATA format card will always have the number of rows and columns punched correctly. Note that since the number of rows and columns of the data matrix is known, when outputting rows via TWOOUT, the header record will also always be correct (see section IV.D.1).

In punching the data and in outputting the data through TWOOUT, the logical variable LOWER, which is the seventh argument in the calling sequences to DWRITE and DMOUT, is checked. If LOWER is true, the upper triangle portion of each row must be completed from the corresponding lower triangle column before the row can be punched or output via TWOOUT. By implication, the LOWER option with value true is used for square symmetric matrices. On printing the data matrix, if LOWER is true, only the lower triangle of the data matrix is printed.

I. FSET

Subroutine FSET is used to set the F format flag on in an output address which will be passed to either DWRITE or DMOUT. The top order halfword of the first argument to DWRITE and DMOUT contains flags which control printing and punching a data matrix (see section IV.H). When a problem program wants a matrix printed in F format rather than the usual E format as the default option, FSET is called with the output address as its only argument. For example, correlation matrices are typically wanted in F format as the default option since all numbers in a correlation matrix are known to range between -1 and 1 and differences between correlations on the order of 10^{-5} are usually considered unimportant. To cause the subroutines DWRITE and DMOUT to print a matrix in F format, FSET turns on the F format flag in the output address.

The point of FSET, however, is that the F format flag is not turned on unconditionally. The F format flag is turned on only if the print flag is on and only if the E format flag is not turned on. That is, the F format flag is turned on only in the case where printed output is requested and only if the user has not specifically requested E format. The bit position for print is masked by the bit pattern PFLAG; the bit position for F format is masked by the bit pattern FFLAG; and the bit position for E format is masked by the bit pattern EFLAG; where PFLAG, FFLAG, and EFLAG are defined for FSET in EQU statements.

J. TPARA

Subroutine TPARA is used by problem programs to read in subparameters, created by the Syntax Interpreter, from DSRN 3. Since there is some variation of subparameter conventions, not all problem programs use TPARA to read subparameters. Those problem programs that do not use TPARA read in their subparameters directly from DSRN 3.

TPARA has three arguments; the array the subparameters are to be read into, the length in full words of the array, and a return address in case of array overflow (i.e. if there are more subparameters to be read than will fit into the first argument array). TPARA reads records from DSRN 3 until either the end of the subparameter string is found or until the array overflow condition is determined. Before each record is read, the end of the anticipated record is calculated and if the address calculated is greater than the end of the memory array, control goes to location TOOBIG. At TOOBIG a return code four is set in register 15 so that upon return to the calling program, control will go to the address specified as the third argument to TPARA. Note that if the array overflow return is executed, the record which would have extended beyond the end of the memory array has not yet been requested and is still available as the next record to be read from DSRN 3.

Each data record consists of a subparameter string and an end of subparameter string indicator, EOSP. EOSP is zero if the current record is not the last one for the problem program, and is non-zero if the current record is the last one for the problem program. The subparameter string within a single record has a length in bytes of LENGTH, where LENGTH has been specified to TPARA in an EQU statement. Note that the end address of each record is calculated in register 4, and that if after the current

record has been read and if EOSP is zero, register 2 is loaded from register 4 and what was the end of the last record is then used as the beginning address of the next record. If a read request results in an end-of-file condition, a SOUPAC system error has occurred; control goes to location EOFADDR and the job is terminated via a CALL ERROR.

K. BCF

Subroutine BCF executes a Branch on Condition instruction for a given condition code and executes a RETURN 1 if the branch succeeded. BCF has three input arguments:

the condition code in INTEGER*4
the first comparand in REAL*8
the second comparand in REAL*8

BCF moves the condition code into a Branch on Condition instruction and then tests the comparands. If either comparand is non-zero, a CDR is executed. If both comparands are algebraically zero, a CLC instruction is executed; this handles the case where one comparand is a true zero but the other is a minus zero. After the appropriate compare is executed, the Branch on Condition is then executed and BCF returns with 0 in register 15 if the branch failed, or BCF returns with 4 in register 15 if the branch succeeded.

V. The Syntax Interpreter

A. General comments

The Syntax Interpreter is run once and only once per SOUPAC job step and is the first program of all SOUPAC job steps to be invoked from the program library by the loader (see section II.A). The Syntax Interpreter is responsible for scanning the PARM field on the EXEC statement which invoked the job step and for reading the user SOUPAC program (i.e. all cards through the END S card). From the information obtained from the EXEC statement and from the user program cards, the Syntax Interpreter creates two parameter files which control the rest of the execution of the job step. The first file is the loading queue, written from subroutine QUEUE using the Queued Sequential Access Method, QSAM. The loading queue is a sequential file with one logical record for each problem program to be invoked by the loader (see section II.A). The second parameter file is the problem program parameter file, written onto DSRN 3. This file provides parameter information which has been extracted from the user program deck to each problem program. The Syntax Interpreter also provides additional information to the loader through use of the SVT.

The scanning algorithms implemented in the Syntax Interpreter are, to a large extent, imbedded in the structure of the Syntax Interpreter and in the relationships between the various Syntax Interpreter subroutines. The subroutines with callable entry points, can be grouped into the following categories for purposes of better understanding the structure of the Syntax Interpreter.

1. The "root" of the Syntax Interpreter "tree."
   a. SEARCH
   b. MAINS
   c. QUEUE

   MAINS is called only from SEARCH; QUEUE is called only from MAINS.

2. Initialization routines called from MAINS
   a. PROLOG
   b. TIOT
   c. During Initialization, PROLOG may call DABTBL, and MAINS may call DADSET. DABTBL and DADSET, although only executed during Syntax Interpreter initialization are actually part of the I/O monitor (see section II.B).

3. OPCODE-the routine which scans the three character mnemonic from a user program card. OPCODE is called only from MAINS, PROLOG, or one of the "subparameter scanning SUB-routines" in group 6 below.

4. Argument scanning control routines.
   a. INTERP, DTERP
   b. LEFT, SKIP
      INTERP is called only from MAINS, PROLOG, or one of the "subparameter scanning SUB-routines" in group 6 below.
      Entry DTERP is called only from MAINS.
      LEFT and entry SKIP are called only from INTERP.

5. Argument scanning routines or "syntax-unit" routines
   a. UNIT
   b. ICONST, FCONST
   c. FORM, TITLE, EBCDIC
   d. INDEX
   e. RELATE
   and argument scanning support routines
   f. ALPHA, LCHK
   g. BETA

6. Subparameter scanning SUB-routines
   a. BALSUB
   b. FRESUB
   c. KCLSUB
   d. LPSUB
   e. MATSUB
   f. PAISUB
   g. PLTSUB
   h. THRSUB
   i. TRASUB
   j. UTLSUB
   k. and more to be added as time goes on.
   These routines are called only from MAINS.

7. Miscellaneous support routines
   a. SCAN, BYE BYE
   b. LAH
   c. SHIFT
   d. CNTROL
   e. SYNTAX, NONE, NO$OPT, ECOUNT
   f. SDUMP, SNEXT, SERROR

Only two subroutines read card images, PROLOG and SCAN. Card images are read into an 80 halfword array in blank common; for scanning, the card image is shifted into a neighboring 80 halfword array, also in blank common. Although these arrays go under various names throughout the Syntax Interpreter, for discussion we shall use the names by which the two arrays are referenced in subroutine SCAN. The first array is CARD, which is the array from which all scanning is done; the second array is CARDUP, the array into which each card image is read. The use of two arrays permits a look-ahead capability at the next card image which will be scanned. The subroutine SHIFT is used to copy the image from CARDUP into CARD.

PROLOG scans card images in a manner unique within the Syntax Interpreter. PROLOG reads into the array CARDUP; determines if the card is a prolog card; and if it is a prolog card, shifts the image into CARD, and then scans it for parameters. When PROLOG reads a card that is not a prolog card, that card image remains in CARDUP and the Syntax Interpreter proceeds with its scanning. This process guarantees that whenever the Syntax Interpreter begins it normal (i.e. post-prolog) scanning, there is a card in the look-ahead buffer ready to be shifted into CARD, the regular scanning buffer. Throughout the rest of the Syntax Interpreter, the individual subroutines assume that the current image to be scanned is in CARD, and all bookkeeping concerned with reading, shifting, and look-ahead information is left to SCAN and LAH. Card image scanning is always done using INTEGER*2 arithmetic.

All parameter information for the running of the Syntax Interpreter is stored in the array IPROG. The following is a list of the various locations in IPROG, the variable names to which these locations are equivalenced, and their purposes.

(IPROG(1),NEXT) - an index value for the current character being scanned within the array CARD.

(IPROG(2),SUB) - a logical variable which is true if parameter scanning is for one of the "SUB-routines" listed in group 6 above.

(IPROG(3),CARDNO) - an integer count of the number in the execution queue of the current problem program being scanned.

(IPROG(4),SUBNO) - an integer count of the current subparameter statement being scanned within the current problem program.

(IPROG(5),EOP) - a logical variable which is true if an END S card is found.

(IPROG(6),EOSP) - a logical variable which is true if an END S card is found or is true if an END P card is found while (IPROG(2), SUB) is true.

(IPROG(7),ABORT) - an integer variable which is increased by one each time an error is found. At the end of execution of the Syntax Interpreter, entry ECOUNT in subroutine SYNTAX is called and the value of ABORT, if non-zero, is printed out, indicating the number of errors found.

(IPROG(8),NUMB) - a logical variable which controls whether the current card image is to be numbered when printed. If the card image is not to be numbered, the current card image is assumed to be an extension of the previous parameter card. NUMB also controls the incrementing of CARDNO for main parameter cards and the incrementing of SUBNO for subparameter cards.

(IPROG(9),NUMFND) - an integer variable which is a count of the number of arguments the Syntax Interpreter has placed in the IPAR array during the scanning of the current main parameter or subparameter card.

(IPROG(10),LNGFMT) - an integer variable which is a length count in single words of a "title" or "format" type argument found by subroutine FORM.

(IPROG(11),FLFLAGS) - a two halfword "array" used to pass F and L print options from subroutine UNIT to MATSUB.

(IPROG(12)PGMNUM) - an integer variable which is the index value into the table of member names contained in MAINS which identifies which problem program is the current program being scanned.

(IPROG(13),SFLAG) - a snapdump flag to be written out in the loading queue record for the problem program currently being scanned.

(IPROG(14),SFFLAG) - an integer count of the number of snapdumps to be executed by problem programs during the job step.

(IPROG(15),CLIMIT) - an integer constant which indicates the number of columns of the user program deck are to be scanned by the Syntax Interpreter for parameter information. The present value assigned to CLIMIT is 80.

(IPROG(16),NPROGS) - an integer count of the number of problem programs which are in the execution queue for the loader to invoke.

(IPROG(17),GALFUP) - an integer variable containing the look-ahead flag.

(IPROG(18),GALF) - an integer variable containing the previous value of GALFUP; GALF indicates the type of card currently being scanned.

(IPROG(19),INPUTC) - an integer variable count of the number of card images read from the user program deck by the Syntax Interpreter.

(IPROG(20),PROL) - a logical variable which is true if parameter scanning is being done for subroutine PROLOG.

(IPROG(21),LCSUSE) - a logical variable which is true if the job step uses Large Capacity Storage. LCSUSE is referenced only from assembly language routines, never from FORTRAN routines.

(IPROG(22),KIND) - an integer variable indicating the type of argument last scanned by subroutine INTERP.

(IPROG(23),NEXTUP) - an integer index value pointing to the first known non-blank character of the card image in the look ahead buffer CARDUP.

Additionally, the following locations in IPAR are important:

(IPAR(24),KOUNT) - an integer count of the number of arguments the Syntax Interpreter has placed in the IPAR array during the scanning of the current subparameter card.

(IPAR(25),LDOLAR) - a flag indicating the type of $ control card found for the current problem program being scanned.

(IPAR(30),LCOPY) - a logical variable which is true if the user included a #COPY control card for the current problem program.

(IPAR(32),LZERO) - a logical variable which is true if the user included a #ZERO control card for the current problem program.

The array EXPARM contains logical variables corresponding to the allowable parameters from the PARM field of the EXEC statement (see Appendix C).

A complete description of all the idiosyncrasies of the Syntax Interpreter is not the intent of this chapter. Intimate knowledge of the inner workings of the Syntax Interpreter can only be achieved by a detailed study of the source code itself. The following informal outline is therefore presented as a beginning guide to the overall program flow of the Syntax Interpreter, and although exceptions exist, this outline is "in general" true.

A. At the MAINS level

1. SEARCH is called from the MAIN macro at the beginning of execution of the Syntax Interpreter. SEARCH scans the PARM field from the EXEC statement and constructs appropriate entries in the array EXPARM in blank common. SEARCH calls MAINS.

2. MAINS performs remaining initializations; calls PROLOG which scans prolog cards (see Appendix D); calls DADSET and TIOT.

3. MAINS scans main program card; calls OPCODE to find value for PGMNUM; calls INTERP to break down remaining parameters on main program card.

4. MAINS process $ control cards, if any. MAINS calls SUB-routines for those programs which have subparameter cards.

5. After processing all parameter cards for a program MAINS calls subroutine QUEUE to write out the loading queue record.

6. MAINS repeats steps 3, 4, and 5 until the END S card is found.

7. MAINS checks to see if any syntax errors were found. If syntax errors were found, the LET variable is checked to see if any programs can be executed or not. MAINS returns to SEARCH.

B. At the "SUB-routine" level

1. Each of the "SUB-routines" (e.g. BALSUB, TRASUB, MATSUB, FRESUB) proceeds to drive the scanning of its respective subparameters.

B. 1. continued:
Since parameter structures differ, different SUB-routines are used. For example, TRASUB allows for statement labels on subparameter statements; BALSUB does not call OPCODE to scan for a mnemonic; the options ONE and TWO in FRESUB are mutually exclusive. LPSUB does not call INTERP to scan parameters for the constraint functions, but rather calls the "syntax-unit" routines directly. Other special case problems cause the SUB-routines to be rather procedure oriented. All SUB-routines return to MAINS.

C. At the INTERP level and the OPCODE level.

1. INTERP receives from its calling program (i.e. MAINS, PROLOG, or one of the "SUB-routines") a table of twenty-four integers which indicate what type of parameters are to be expected on the current card. INTERP calls LEFT which skips down the card until it finds the next left delimiter; INTERP then calls the appropriate "syntax-unit" routine. If LEFT finds as a left delimiter the character ".", INTERP terminates the parameter scan and returns to its caller. Note that INTERP does not need to call SCAN directly.

2. From the INTERP and OPCODE level on down through the SCAN level, a program may return to its caller either normally or via a RETURN 1. RETURN 1 means that an END P or an END S card has been found. (The exception is LEFT which uses RETURN 1 to indicate that a period has been found; RETURN 2 indicates that an END P or an END S card has been found.) By this RETURN 1 mechanism, the fact that an END P or END S card has been found is passed back up the syntax tree and appropriate action can be taken at any given level.

   For example, finding an END P or END S card is an error in all cases except in the case where SCAN is being called for the first time for a new parameter card. In all other (i.e. error) cases, it is desirable that the scanning process stop cold at all levels below the calling program to INTERP or OPCODE; hence the RETURN 1 route bypasses all additional parameter scanning by routines from INTERP and OPCODE on down.

D. Error Procedures

1. Whenever an error is found, ABORT is increased by one, and an error message is printed out.

2. Errors are always signalled at the lowest possible level. This prevents the identical error to be signaled as an error at more than one level. For example, suppose the current hierarchy of calling programs is:

MAINS
MATSUB
INTERP
UNIT
SCAN

2. continued:

and an END P card is found. Finding an END P card is not an error to SCAN, therefore SCAN merely executes a RETURN 1. Finding an END P card is, however, an error to UNIT; in general, finding an END P card while any "syntax-unit" program is in the control tree is an error, and all "syntax-unit" routines call NONE to signal this fact. NONE will increase ABORT by one and print out an appropriate error message. The "syntax-unit" routine will then execute a RETURN 1 to its caller, in this case INTERP.

Being returned to via a RETURN 1 represents errors to both INTERP and MATSUB. Neither will signal an error, however, since errors are always known to be signalled at the lowest possible level.

When a syntax error is found by a "syntax-unit" routine, an error is signalled by a call to SYNTAX. At entry SYNTAX, ABORT is increased by one and an appropriate error message is printed out. SYNTAX returns normally to the "syntax-unit" routine, and the "syntax-unit" routine returns to its caller, with the "syntax-unit" routine's caller none the wiser that an error was signalled.

3. When ABORT is found to be non-zero, no additional parameters are written onto DSRN 3 nor are loading queue records written out.

4. Besides errors, there are two other classifications of exceptional conditions which are signalled by the Syntax Interpreter; warnings and super errors. Warnings are simply messages which inform the user that some exceptional condition has been found. Warnings do not have any other immediate effect. Super errors result in an immediate termination of the execution of the Syntax Interpreter. Super errors always exit through entry BYE BYE in subroutine SCAN.

5. Upon termination of the execution of the Syntax Interpreter either at the end of MAINS or at entry BYE BYE, entry ECOUNT is called to print out a final count of the number of errors found by the Syntax Interpreter (i.e., the value of ABORT).

B. SEARCH

Subroutine SEARCH is called from the MAIN macro at the beginning of execution of the Syntax Interpreter, and calls the termination routine SNEXT at the end of execution of the Syntax Interpreter. SEARCH is also responsible for determining the current date from the OS Communications Vector Table, the CVT; printing out the job step message WELCOME TO SOUPAC; scanning the parameters from the PARM field of the EXEC card which invoked the SOUPAC job step; printing out the SOUPAC OPTIONS and current date; opening and closing the loading queue data set PGMQUEUE whenever necessary; calling MAINS; and moving parameter information into the SVT in the loader.

Upon entry to SEARCH, the contents of APGM in the SVT is copied into the second fullword of the named common area SSPACE. APGM at this time contains the address of the DCB for the program library SOUPLIB (see section III.B). Next, the current date is extracted in packed decimal form from the OS/360 CVT and converted to the form used at the top of each page of the user's program listing. An index value for the current month, to be used in referencing the SOUPLOG activity file, is stored in the halfword at APGM+2 in the SVT. The symbolic data as constructed from the packed decimal form in the CVT is copied into entry THEDATE in subroutine CNTROL for use in printing at the top of each page of the user's program listing. The heading WELCOME TO SOUPAC is printed out.

After the heading has been printed out, the PARM field from the EXEC statement is scanned. Location PAUL in the SVT at this time contains the address of a fullword which contains the address of the halfword on a halfword boundary which is the argument string constructed from the EXEC statement PARM field. The first halfword, on a halfword

boundary, contains a count of the number of characters which were in the PARM field on the EXEC statement. If this count halfword is zero, the parameter scanning is skipped and control goes to location LISTSET. If the count halfword is non-zero, the character string, beginning immediately after the halfword count, is compared against the legal job step options allowed by SEARCH. If more than one option is specified, additional options must be preceded by a comma. If a non-legal character string is found, a warning message is printed, the remainder of the character string is ignored, and control goes to location LISTSET. Appendix C contains a description of the legal options allowed on the EXEC card.

Each option to a SOUPAC job step has two forms; either the option name itself (e.g. LET), or the option preceded by the characters NO (e.g. NOLET). Only one of these two forms of a given option may appear in the PARM field, and no option may appear more than once. Corresponding to each option and its alternative NO form is a location in the eight word array EXPARM in blank common. For a given option the corresponding location in EXPARAM is true if the option is not preceded by the characters NO (e.g. LET implies true) and is false if the option is preceded by NO (e.g. NOLET implies false). Default values are assigned to the EXPARM variables unless overridden by the EXEC statement PARM field.

At location LISTSET the table of default values with changes, if any, as a result of the EXEC statement PARM field, is scanned and the SOUPAC OPTIONS= print line is constructed and then written out. Note that the date has already been inserted into the correct location at the far right of the print line.

The job step option PGM (see Appendix C) is checked for the default value true. If NOPGM was specified, the Syntax Interpreter does not have to scan a user program since use of NOPGM as an option implies that the user is running from parameter and loading queue files created previously by the Syntax Interpreter. If the logical value for PGM is true, the DCB for the loading queue PGMQUEUE is opened for output. The DCB for the loading queue is an entry point to SEARCH so that it can be referenced by subroutine QUEUE. Finally, MAINS is called and the scanning of the user program cards are scanned through the END S card.

After return from MAINS, the logical value for PGM is checked again. If the value for PGM is true, control goes to location SKIP2 where the DCB for the loading queue is closed. If the value for PGM is false and the value for EXECUTE is false, control goes to location COUNTSET. If the value for PGM is true and the value for EXECUTE is true, the user provided loading queue is opened and read to determine how many loading queue logical records are present and consequently the number of problem programs to be invoked by the loader for the job step. Control then falls to location SKIP2 where the loading queue data set is closed.

A check is then made to see if NOEXECUTE was specified. If NOEXECUTE was specified, the count of the number of problem programs to be invoked by the loader is set to zero. At COUNTSET, a test is made to see if the number of programs to be invoked by the loader is zero. If the number is zero, the Syntax Interpreter is the only, and hence last, program to be invoked by the loader. This fact is indicated to subroutine NEXT and the loader by setting location ENDFLAG in the SVT to one. The number of programs to be invoked by the loader and a count of snapdumps to be executed are shifted into the SVT. Subroutine SNEXT

is called to terminate the execution of the parameter scanning by the Syntax Interpreter.

C.1. MAINS

MAINS is called from subroutine SEARCH, and is the principal driving program for the rest of the Syntax Interpreter. Besides the usual tables for the parameters used in calling OPCODE and INTERP, MAINS also contains the following additional tables:

a table of member names for all problem programs in the SOUPAC system
complete EQUIVALENCE statements for reference of all equivalences in the arrays IPAR, EXPARM, and IPROG.

Upon entry to MAINS, an initialization of the Syntax Interpreter is executed. IPAR and IPROG are initialized; DSRN 3 is written onto so that control blocks are allocated in regular memory for DSRN 3 by the I/O monitor routine FIOCS#. A call is made to PROLOG, and if no errors were found during execution of PROLOG, DADSET and TIOT are also called. This is the only place in the Syntax Interpreter that calls PROLOG, DADSET, and TIOT. After initialization is complete, a check is made of the logical variable PGM. PGM is in the array EXPARM and is set by subroutine SEARCH (see section V.B). If PGM is true, normal execution of MAINS continues. If PGM is false (i.e. the user coded NOPGM in the PARM field on the EXEC card), the END S card is checked by a call to SCAN. If the END S card is present, control goes to statement 9900, otherwise the super error exit BYE BYE, an entry in subroutine SCAN, is called to terminate execution of the Syntax Interpreter.

If PGM was true, normal processing by the Syntax Interpreter continues. MAINS next sets NUMB to true, indicating that the next card scanned is assumed to begin a new SOUPAC statement, and therefore it should be numbered.

SFLAG is set to zero, indicating no snapdump has been indicated for the program to be scanned and IPAR(25) through IPAR (32) is set to zero indicating that no $ control cards have been encountered for the current problem program. All elements of LUNIT are set false, indicating that no unit addresses have been encountered for the current program. As unit addresses are encountered in subroutine UNIT, the LUNIT entry corresponding to the DSRN for the address will be set true. Next, SCAN is called, and if an END P or END S card is found, control goes to statement 7. If the card was an END S card, control goes to statement 9900 for final checkout before MAINS returns to SEARCH.

If SCAN does not encounter an END S card, subroutine OPCODE is called to see which three character mnemonic was used (i.e. which program the user wants to be invoked). The number of the current program is returned in PGMNUM. After OPCODE, INTERP is called to scan the parameters for the main parameter card. A check is made to see if the current program allows $ - cards or subparameter cards. If scanning of $ - cards is required, that is done by calling entry DTERP in subroutine INTERP. If subparameter cards are to be scanned the appropriate "SUB-routine" is called. If no errors have occurred, NPROGS is increased by 1, QUEUE is called to write out the membername, and control goes back to statement 5 where NUMB is set true. This entire process repeats until an END S card is found or unless a super error somewhere directly terminates execution of the Syntax Interpreter.

When an END S card is found, control goes to statement 9900. At 9900 the logical variable CRITA is checked to see if the version of the Syntax Interpreter is within the actual SOUPAC system or SCANSOUP.

(SCANSOUP is used to perform a syntax check of a SOUPAC program deck under an Express system). If the Syntax Interpreter is executing within the actual SOUPAC system, ECOUNT is called to print out an error summary if any syntax errors were found. If any errors were found, a check is then made to see if LET was specified as a parameter on the EXEC card (see Appendix C). If NOLET was specified, NPROGS is set to zero. Otherwise, the logical variable PGM is checked and if PGM is true, a REWIND is executed for DSRN 3, the problem program parameter file. MAINS then returns to SEARCH.

C.2. QUEUE

Subroutine QUEUE creates the loading queue for the remainder of the job step. In creating the loading queue, QUEUE writes onto the file referenced by the ddname PGMQUEUE one record for each problem program the loader is to invoke (see sections II.A and II.C). The actual DCB for PGMQUEUE is in SEARCH and is opened and closed by SEARCH. SEARCH stores the address of the DCB for PGMQUEUE in location NAMES. NAMES is an external reference in SEARCH set up specifically so that QUEUE can get the address of the DCB (see section V.B).

QUEUE is called from MAINS once for each problem program scanned by MAINS until either an error is found or the END S card is read. QUEUE is passed the member name to be written onto the loading queue as an eight character argument.

Before writing the member name onto the queue, QUEUE executes a BLDL macro instruction on the member name argument. Returned as part of the BLDL is the size of the program with the corresponding member name. QUEUE then proceeds to make a preliminary check as to whether or not there will be enough memory available for the given problem program to execute when its turn comes.

The loading queue records which QUEUE finally constructs are 20 bytes long and consist of:

The eight character member name

a four byte snapdump flag indicating what type of snapdump, if any, is to be executed when that particular problem program terminates (see section IV.B).

a four byte memory flag which is a count of the number of bytes to be reserved by ALLOC8 for system use. As the problem program is running this word is shifted into MEMORY in the SVT by the loader (see sections II.A, IV.B, and III.C).

a four byte word containing the number appropriate for reference of a record number in the SOUPLOG data set. This number is saved by the loader in the low order halfword of APGM in the SVT (see section II.A).

C.3. PROLOG

PROLOG is called by MAINS at the beginning of execution of MAINS. The purpose of this subroutine is to scan any prolog cards at the beginning of the user deck (see Appendix D). As soon as PROLOG reads a card which is not a prolog card, PROLOG returns to MAINS. The logical variable PROL in IPROG is set to true by MAINS immediately before MAINS calls PROLOG and is set to false immediately after PROLOG returns to MAINS. PROL remains false for the remainder of execution of the Syntax Interpreter.

PROLOG has one argument which is 24 halfwords long. This array is filled in by PROLOG if a #TEST prolog is included. This array is used to define the argument types for executing programs with the mnemonic TEST.

C.4. TIOT

TIOT is called from MAINS and is called only once per job step. Upon entry to TIOT, the Program Queue Elements are checked to find if LCS is available to the job step, how much region is available to the job step, and the size of the Syntax Interpreter. The region size information is used later by subroutine QUEUE (see section V.C.2).

The major function of TIOT is to allocate a Unit Block, a Data Control Block, and two Data Event Control Blocks, a total of 164 bytes, for each FTnnFnnn type DD card. The address of the job step TIOT is found via the CVT (Communications Vector Table) and the TCB (Task Control Block). Each TIOT entry is checked for DD name type. If the DD name does not begin with FT , allocation is skipped. If LCS bulk storage memory has been allocated to the job step, these control blocks are placed in LCS. The essential reason for the TIOT subroutine is to avoid the memory fragmentation which would otherwise result if the control blocks for each file were allocated the first time that file was used. Note that the allocation procedure must also make appropriate entries in the FORTRAN I/O Monitor Unit Assignment Table so that FIOCS# knows that the control blocks have already been allocated.

TIOT also writes memory size information into the log dataset SOUPLOG. For this reason TIOT is in the form of a conditional assembly (i.e., macro) in the same way that subroutine NEXT is. If SOUPLOG is to be used, the parameter ACTIVITY is coded on the TIOT macro call; if ACTIVITY is absent no TIOT records are kept in SOUPLOG.

C.5. MESS

This subroutine is called by MAINS as one of the first functions MAINS performs. The sole purpose of this subroutine is to provide information to all users by printing out any messages about the status of SOUPAC. The actual message, therefore, changes from time to time. If there is no message to be printed out, MESS returns directly to MAINS; if there is a message, MESS prints it and then returns to MAINS.

D.1. OPCODE

Subroutine OPCODE is called to identify the mnemonic for the current main parameter card or subparameter card. OPCODE is passed a table of permissible mnemonics and a count of the number of entries in the table; the table is referenced as TNAMES within OPCODE, and the count as IN. In return, OPCODE performs a sequential search through TNAMES comparing entires in TNAMES against the contents of the mnemonic candidate, constructed in the local array TEST. If OPCODE succeeds in identifying the mnemonic, an integer K is returned to the calling program indicating which entry in TNAMES succeeded. If OPCODE fails, an error message is printed, ABORT is increased by one, and OPCODE, returns via a RETURN 2, unless five consecutive invalid mnemonics are found in which case a super error is signalled and SERROR is called to terminate parameter scanning. OPCODE signals an error and executes a RETURN 1 if an END P or END S is found before a mnemonic is successfully found.

All tables passed to OPCODE as TNAMES have four characters per mnemonic, however OPCODE only considers the first three of them. This construction was adopted so that the mnemonic search could be trivially expanded to a four character search, if desired, by redefining the value of the variable LOOK. For certain operations, such as IF in TRANSFORMATIONS, it is desirable to have only a two character search. As a result, OPCODE, when searching TNAMES, will succeed when a blank is found in TNAMES implying that the non-blank characters of the TNAMES entry has been matched. For this reason, it is important that within a given table passed as TNAMES, no three character mnemonic have its first two characters the same as some two character mnemonic.

The characters as scanned are copied into TEST. Testing is done out of TEST rather than directly from CARD since it is necessary to always have all scanned characters available. For example, suppose in MATRIX, the user specifies

the ABS operation. Within the table passed by MATSUB to OPCODE, the entry for ADD precedes ABS. The A of ABS will succeed as the first letter of ADD, however the second letter will fail. However, we must somehow have available the letter A for correct identification of ABS. If the mnemonic is split across two card images, unlikely but legal, the letter A would otherwise be lost after scanning subsequent characters if the letter were not saved in TEST.

After OPCODE has successfully identified a particular mnemonic, the subroutine executes a normal return to its calling program, returning in K the number of the OPCODE within TNAMES.

D.2. INTERP, DTERP

Subroutine INTERP, with entry DTERP, is used to drive the process of scanning parameters by the "syntax-unit" routines. This subroutine is called from MAINS, PROLOG, and the various SUB-routines. Entry DTERP is called only from MAINS specifically for the purpose of scanning $ control cards.

INTERP is table driven in nature; the parameters which it chooses to scan are determined by a 24 halfword table of integers, referenced within INTERP under the name OPLIST. The integers passed to INTERP are from large tables kept in each of the calling programs. Individual table entries have permissible values ranging from 0 to 9. For partially historical reasons, the first integer of a given 24 integer list is always zero. INTERP indexes across the OPLIST argument, therefore, beginning with 2 and ending at 23. Note that the choice of 23 as a stopping point is again an arbitrary design decision which was made in deference to the 7094 SSUPAC system precedent. Parameters, as scanned, go into the IPAR location which is in the same relative position as its OPLIST entry. For example, if OPLIST(3) indicates that a fixed point constant is to be scanned, the constant when scanned will be placed in IPAR(3). Integers in OPLIST(2) through OPLIST(23) have the following meanings:

0	No parameter
1	Address, e.g., CARDS, PRINT,S1
2	Fixed point constant
3	Floating point constant
4	If scanning for the MATRIX program, 4 means either an address or a floating point constant, depending upon the first character of the parameter. In all other instances, 4 means that the parameter is either a fixed point constant or a floating point constant, again depending upon the first character

5	Format
6	Relational operator, e.g., "NE"
7	If scanning for TRANSFORMATIONS, this argument means scan an 8 byte character argument. Otherwise (MATRIX program case), scan a character string with maximum allowable length of 128 bytes.
8	Index set argument
9	End of parameters to be placed in IPAR. Go to OPLIST(24) to determine what type of parameters to be scanned for JPAR.

Parameter types 1, 2, 3, 4, and 6 cause the scanned argument to be placed in the IPAR position corresponding to the OPLIST position of the argument. Parameter types 5 and 7 cause the length of the scanned parameter to be placed in IPAR; the actual format or character string has been placed in the common array FMT (see section V.E.3). Note that no program may have both a format argument and a 128 byte maximum length character string since they are both stored in FMT. There is no provision for more than one argument at a time stored in FMT.

OPLIST(24) is never checked unless the last non-zero entry in OPLIST(3) through OPLIST(23) was a 9. OPLIST(24) can have any of 9 possible values 0 through 8. The zero value is not of interest since OPLIST(24) is 0 if and only if the last non-zero entry in OPLIST(2) through OPLIST(23) was not a 9. The remaining possible values for OPLIST(24) have the following meanings:

1	Address
2	Fixed point constant
3	Floating point constant
4	Fixed or floating point constant; for the MATRIX program, only floating point constants allowed.
5	not in use
6	not in use
7	8 byte character string
8	Index set

When scanning parameters for OPLIST(24), scanning continues until the end of the complete parameter statement, i.e., a terminating period, is found. In principle, an indeterminate number of parameters of the same type may be scanned.

Parameters scanned under control of OPLIST(24) are stored in consecutive locations in JPAR rather than IPAR. Note that each 8 byte character argument takes two locations in JPAR, and each index set argument takes three locations. Note that there is no provision for scanning an indeterminate number of "long" (i.e., format or long character string) arguments.

INTERP, when finished, returns to its calling program. At the time of its return, NUMFND contains the index of the highest IPAR location filled; LNGFMT contains the length of any string argument in FMT. (LNGFMT is set by subroutine FORM); and KOUNT, in IPAR(24) contains the highest JPAR location filled. The one error condition which INTERP can identify is if an OPLIST entry is zero, but the user has remaining unscanned parameters on his current parameter statement.

An understanding of the structure implied by INTERP is of critical importance in understanding of the Syntax Interpreter. The original design intention was to provide a consistent framework within which parameters, parameters modeled after the 7094 SSUPAC system, could be scanned. What was desired was a table driven procedure to supervise the whole operation. In particular, the desire was to avoid ad hoc code for unique situations. For the "simple" syntax which existed in the 7094 SSUPAC system at the time INTERP was written, this goal was easily realized.

The problem which immediately arose, however, was that as new ideas were injected into SOUPAC, there was the problem of representing these new features in the source language while still retaining what was being used. Of necessity, ad hoc code was written. For example, the RECODE instruction was so "flexible" in design as to completely transcend anything INTERP could handle. A separate subroutine had to be written to handle that one instruction. Furthermore, the restrictions of INTERP, and the original SSUPAC syntax, required some rather bizarre keypunching antics to be performed in order to keypunch often repeated lexical structures, for example, arguments enclosed in parentheses. Ah, the trusting innocence of youth.

D.3. RECODE

This subroutine is called only from TRASUB and is used to supervise the scanning of parameters for the RECODE operation. This subroutine is essentially a substitute for INTERP for the RECODE statement. The reason that RECODE was needed was that after a "condition set" is scanned, condition set being a pair of index sets separated by a relational operator, a decision must be made as to whether this is a logical connective (e.g., "AND", "OR") and another condition set follows, or whether the final recode sets follow. This decision can only be made on the basis of the leading delimiter, a quote or not a quote, of the following parameter. An OPLIST type table would not convey this information readily (see section V.D.2).

D.4. LEFT, SKIP

Subroutine LEFT searches for left delimiters. It is called by any subroutine which is about to call a "syntax-unit" routine. When a left delimiter is found, the presumption is that it signals the beginning of a new argument. The set of left delimiters is:

* " (.

The period character is special in that if a period is found as a left delimiter, this indicates the end of the parameter string. As a result, if a period is found, LEFT executes a RETURN 1 to signal this fact. If an ENDP or ENDS card is found as a left delimiter, an error is signalled (i.e. the message MISSING TERMINAL DELIMITER, meaning missing period, is printed), and a RETURN 2 is executed. If while looking for left delimiters a right parenthesis is found, a warning message is printed, and scanning for a left delimiter continues. When the left delimiter is found, LEFT executes a normal return. The normal return leaves the variable NEXT pointing to the delimiter within the array CARD so that the appropriate "syntax-unit" routine (e.g. UNIT, ICONST, FCONST, RELATE, etc.) when called can check that the delimiter is correct for that type of argument.

Entry SKIP is called to skip to the end of the parameter card. SKIP is called only under certain error conditions. SKIP scans characters until it finds a period which it considers to be the period terminating the parameter statement, and then it returns. SKIP ignores periods which it determines are within asteriske (i.e. ignores what it hopes are floating point constants), and within quotation marks (i.e. ignores what it hopes is a character string or format). If SKIP finds an ENDP or ENDS card before it finds a period, an error is signalled, and a RETURN 1 is executed.

E.1. UNIT

Subroutine UNIT is called whenever it is necessary to scan an address argument. Address arguments are enclosed in parentheses and consist of information from the following types:

file address information -- C; S1 through S40; T1 through T40; D1 through D40; V1 through V9; or I
print information -- P; P(E); P(F); P(E,L); P(F,L); P(L,E); or P(L,F)
punch information -- X

If more than one type of information is desired for a given matrix, the information types may appear in any order and are separated from each other by slashes. For example, the following addresses are equivalent:

(S1/P/X)	(P/X/S1)
(S1/X/P)	(X/S1/P)
(P/S1/X)	(X/P/S1)

Also, no one type of information may appear more than once in a given argument. For example, the following addresses are illegal and will be flagged as errors by UNIT:

(S1/S2) (P/S1/P(F)) (S1/T3)

Under file address information, note that S1 through S40 and T1 through T40 are equivalent and correspond to DSRN's 11 through 50; D1 through D40 correspond to DSRN's 51 through 90; C stands for card input; I stands for the INCORE address option available in MATRIX.

UNIT returns its result in the argument ADDR. ADDR is treated as two halfwords, with the highorder halfword containing print, punch, E format and F format information, and the loworder halfword containing the file address. S, T and D file addresses are converted to their corresponding DSRN's.

The address C is converted to address 5, and the address I is converted to the address 500. Print, punch, and format information is indicated by various bits in the highorder halfword. The address created by UNIT is appropriate for use directly as the address argument to the routines:

DREAD	DMIN
DWRITE	DMOUT

ROWIN also accepts these addresses but does not honor print or punch.

Note that for S, T, or D unit addresses, the corresponding logical variable in array LUNIT is set true for that DSRN. In this manner, when QUEUE is called by MAINS, LUNIT will have a true value for each DSRN used by the current problem program being scanned. This information will be used by QUEUE to help determine region requirements for the problem program.

Another function which UNIT performs is resolving V unit addresses. As UNIT recognizes a V unit address, it secures the corresponding address from tables created by PROLOG, and substitutes the correct value into ADDR.

E.2. ICONST, FCONST

Subroutine ICONST is called whenever it is necessary to scan a fixed point constant argument from a parameter card. Upon entry to ICONST, the left delimiter is checked to see if it is a left parenthesis or a comma, and if the left delimiter is neither, an error is indicated. At this point, subroutine ALPHA is called to get the actual fixed point constant. After return from ALPHA if the next character is a valid right delimiter, a normal return is executed. If the next character is not a valid right delimiter an error is indicated.

Entry FCONST is called whenever a floating point argument is to be scanned from a parameter card. Upon entry to FCONST, the left delimiter is checked to see if it is an asterisk, and if it is not, an error is indicated. If the left delimiter is an asterisk, BETA is called to get the actual floating point constant. Upon return from BETA, the right delimiter is checked to see if it is an asterisk. If it is not an asterisk, an error is indicated, otherwise a normal return is executed.

E.3. FORM, TITLE, EBCDIC

FORM and TITLE are used to scan formats and arbitary character strings, respectively. Both use quotation marks as delimiters. Additionally, formats must begin with a left parenthesis, end with a right parenthesis, and have a matched number of parentheses. Formats scanned by FORM can be a maximum of 592 characters long, and character strings scanned by TITLE can be a maximum of 128 characters long. The resulting argument scanned by either FORM or TITLE is placed into the FMT array in blank common. Its length is returned in the calling argument J and stored in LNGFMT equivalenced to IPROG(10).

Entry EBCDIC is used to scan eight character labels which are set off by quotation marks. EBCDIC returns its result in the eight character calling argument WHACKO. Note that all three routines scan arguments out of CARD which has one character every two bytes and pack their results into an answer, in either FMT or WHACKO, in one character per byte.

E.4. INDEX

Subroutine INDEX is called whenever an index set argument must be scanned from a parameter card. In general, index set arguments are fixed point sets but in the case of TRANSFORMATIONS, they can be either fixed or floating point sets. Upon entry to INDEX, the left delimiter is checked to see if it is a left parenthesis. If it is a left parenthesis, a fixed point set is scanned. If it is not a left parenthesis, a check is made to see if it is an asterisk and if the current problem program is TRANSFORMATIONS, in which case a floating point index set is scanned.

When scanning fixed point index sets, subroutine ALPHA is called to scan the actual integers in the set. Arguments within the set are separated by commas, and a maximum of three arguments is allowed within the set. If the current program is not TRANSFORMATIONS, a check is made to make sure that the fixed point index set arguments are not less than or equal to zero; if the check shows an error, a message is printed and ABORT is increased by one. TRANSFORMATIONS index sets are, in general, checked for less than or equal to zero or greater than two thousand; exception, the CONSTANT operation in TRANSFORMATIONS. In the case of a fixed point index set while scanning a TRANSFORMATIONS card, DO-FLAG and FLAG-DO notations are also handled.

When scanning floating point index sets, BETA is called to scan the actual floating point argument. Arguments within the set are separated by commas, and a maximum of three arguments is allowed.

Examples of index sets allowed under various conditions:

(4) (1,5) (1,10,2) (201F,202F) *1.,3.,1.*

Note that the last two examples are allowed only in TRANSFORMATIONS.

E.5. RELATE, CONNEC

Subroutine RELATE scans syntax units of the set

"GT" "LT" "NE" "EQ" "GE" "LE"

Note that the left and right delimiters for the relational operators is the quotation mark.

RELATE returns two arguments, CCODE and NTYPE to its caller. Each argument is an encoding for the particular relational operator, there are simply two distinct encodings used. The CCODE encoding is a condition code which can be used directly in an assembly language Branch instruction as provided by BCF (see section IV.K). MATRIX and TRANSFORMATIONS use the CCODE encoding for their uses of relational operators. The NTYPE encoding is a 1,2,3 encoding for the operators, GT, LT, and NE respectively and is used exclusively by LINEAR PROGRAMMING.

Entry CONNEC scans syntax units of the set

"AN" "OR" "EO"

CONNEC is called by subroutine RECODE for the purpose of scanning the TRANSFORMATIONS RECODE instruction. For entry CONNEC, the two return argument encodings CCODE and NTYPE are identically 1,2,3 for the three connectives.

E.6. ALPHA, LCHK

Subroutine ALPHA is called whenever it is necessary to scan an integer as part of some type of "syntax-unit" scan. ALPHA looks only at characters in the set of integers 0 through 9. As soon as ALPHA encounters a character other than one in the set 0 through 9, it returns to its caller. ALPHA is not concerned with the question of whether or not there is a syntax error; ALPHA does not have enough information to make that judgement. It is rather the responsibility of its caller to determine whether or not the particular integer argument is set off by proper delimiters.

Entry LCHK is used to deal specifically with labels within the TRANSFORMATIONS program of the type used in

IF (2) "*+1" "*+3" "*+5".

LCHK determines the value of the integer after the asterisk and adds that number to the number of the current suboperation.

E.7. BETA

Subroutine BETA is called by either FCONST or INDEX whenever a floating point constant is to be scanned. BETA scans constants of any of the type:

x	x E z
x.	x. E z
.y	.y E z
x.y	x.y E z

where x,y, and z are decimal numbers, and x and z can optionally have either a leading plus or a leading minus sign. The " x.y E " portion of the floating point constant is scanned by BETA directly. To scan the " z " portion, BETA calls subroutine ALPHA.

BETA is unconcerned with delimiters in any syntax error sense. BETA scans a floating point constant, if it can, and returns to its caller when it encounters a character it does not recognize. BETA does, however, execute a RETURN 1 whenever it is returned to from SCAN or ALPHA by a RETURN 1.

F. The SUB routines

Each problem program which requires subparameters to be scanned must be supported in the Syntax Interpreter by a SUB routine. Programs of this type are TRANSFORMATIONS, MATRIX, BALANOVA, REGRESSION, FREQUENCY, K-CLASS, and others. To scan subparameters for TRANSFORMATIONS is TRASUB, for MATRIX is MATSUB, for BALANOVA is BALSUB, and so on. As MAINS finds a main parameter card for a program which has subparameter statements, MAINS calls the appropriate SUB routine to look for the subparameters. Within the load module SEARCH, the SUB routines are overlayed against each other since only one such routine is ever needed in memory at a time.

The necessity of having separate routines for each program with subparameter cards is that there are slight differences in the rules for each of the programs which had to be taken into account. For example, several programs, notably BALANOVA and K-CLASS, do not have mnemonics which begin the subparameter statements since these programs each have subparameters of only one type. TRANSFORMATIONS must handle the resolution of statement labels for branching purposes. It must recognize multiply defined or undefined labels, and signal an error if a branch is indicated which branches across the LAST suboperation card. Note the essential two pass nature of the label processing feature. To handle this, TRASUB uses FT02F001 as a temporary storage area. After all TRANSFORMATIONS statements have been read, FT02F001 is reread, and label references are resolved. The MATRIX program likewise has pecularities of its own, largely because MATRIX and MATSUB were written before some of the common conventions were established.

The SUB routines, in general, have the same type of mnemonic tables and tables for INTERP as MAINS has.

We have, therefore, the situation that as more programs are added to SOUPAC, those programs which require subparameters must have their own SUB routine specially coded. This is especially difficult in that newer programs tend to be more sophisticated and, as a result, require this individual attention. The addition of programs which do not require subparameters, by contrast, require merely simple table additions to MAINS.

In general, the array XARRAY in the named common area SHARE is used by the various SUB routines for constructing the subparameters for the individual problem programs. As XARRAY becomes filled, and also at the end of execution of each SUB routine, XARRAY is written out onto unit 3.

A few special comments are in order.

TRASUB

As previously mentioned, one of the major differences in TRASUB from the other SUB routines is the problems involved with label referencing. At the beginning of each new subparameter statement, the first character is checked to see if it is a quote mark. If it is a quote mark, EBCDIC is called to scan the label, and the resulting 8 byte character string is returned to TRASUB. If the label has been previously defined, an error is signalled. If the label has not been previously defined, it is entered into the table LABELS in the named common area TRACOM, and the count of labels found is incremented by one.

At this point, OPCODE is called to find the mnemonic of the instruction. All subroutines within the Syntax Interpreter perform a call SCAN immediately prior to a CALL OPCODE, and TRASUB is no exception. Why, then, wasn't CALL SCAN simply made the first statement of OPCODE with the corresponding burden lifted from all of OPCODE's caller? The answer lies in the necessity of TRASUB's looking at that first character of a statement to determine whether or not it is a quote mark and hence the beginning of a label instead of the beginning of a mnemonic. TRASUB must call SCAN at least once before calling OPCODE to get a pointer to that first character.

After calling OPCODE, TRASUB verifies that LAST has not been used twice within one program. Next, either INTERP or RECODE is called to find the actual subparameters. As buffers of parameters fill up, the COVER array, containing flag and type information for each of the operands in XARRAY, is written onto the program parameter file on unit 3. The actual parameters are temporarily written onto FT02F001. After the ENDP card is found, the program parameters are read from FT02F001, label references are resolved using the table BRANCH, and the parameters are written onto unit 3, FT03F001.

The table BRANCH is a three column array which is constructed during execution of TRASUB to determine where label references have been used. Each time a label reference occurs as an operand, an entry is made in the next available place in the first column of BRANCH. This entry points to the location in the complete XARRAY which corresponds to the occurrence of that label usage. The second column points to the actual 8 byte label in the array LABELS. This second column entry is filled by INTERP as each

label is found. The third column entry is the "resolution" entry which indicates the statement number of the statement having the label indicated in the second column entry. The first column simply points to the location in XARRAY that the statement number is to be placed as the subparameters are read back from FT02F001 and copied onto FT03F001. An undefined label is found whenever there is a third column entry which is zero.

BALSUB

BALSUB is typical of those SUB routines which scan subparameter cards which do not have mnemonics. Normally, when SCAN is called to point to the first non-blank character of a statement, if that "statement" is really an ENDP card, SCAN will return with a RETURN 1, and scanning of additional cards is stopped. For subparameter cards without mnemonics, this luxury is not available, since INTERP is called directly, and the first character, a delimiter, must still be available. The trick used in place of the RETURN 1 option from SCAN is to explicitly check the look-ahead flag GALFUP to see if the card in CARDUP is an ENDP or ENDS card. If CARDUP does contain a card of that type, subparameter scanning is terminated. Control goes to the end of BALSUB, then SCAN is called to actually scan the ENDP or ENDS card.

G.1. SCAN, BYE BYE

Subroutine SCAN is one of the key routines in the running of the Syntax Interpreter. The primary function of SCAN is to set the variable NEXT to point to the next non-blank character in the CARD array. As each (non-blank) character from the input control cards is required, the particular subroutine which needs to cause the pointer NEXT to be advanced calls SCAN. On normal return, NEXT points to the next non-blank character. On RETURN 1 condition, SCAN has found either an ENDP or an ENDS card.

In the performance of this simple function, SCAN also

1. Calls SHIFT to shift cards from the look-ahead area CARDUP into CARD.
2. Reads new cards into CARDUP, except if an ENDS card has been found. For new cards read SCAN calls subroutine LAH to identify the type of card read (see section V.G.2).
3. Prints out the card images with program or suboperation numbers and with indentation as appropriate.
4. Processes all # control cards which are not prolog cards (e.g., #SREP, #EREP, #LIST, #SNAP).
5. Handles all processing of repeat sequences. As repeat sequences are encountered, the card images are written onto FT01F001. When the repeat sequence has been terminated by a #EREP card, the contents of FT01F001 are read back an appropriate number of times and processed as ordinary card images. When card images which were read back from FT01F001 are printed out, they are preceded by a + character to identify them as coming from a repeat sequence.
6. Processes ENDP and ENDS cards by executing a RETURN1.

Entry BYE BYE is used as a common exit point for all SUPER ERROR exits. Its position in SCAN is simply the convenience of scan as being a permanently resident part of the root segment which is also at the lowest level of the Syntax Interpreter program "tree."

G.2. LAH

Subroutine LAH is used to provide look-ahead information on the next card image to be scanned. LAH checks the card image currently in the look-ahead buffer, CARDUP, and assigns a value to the look-ahead GALFUP according to the following table:

GALFUP	Type of card image in CARDUP
1	blank card
2	$ control card
3	# card or zilch card
4	ENDP card
5	ENDS card
6	DATA format - SUPER ERROR condition

If the image in CARDUP is not one of these six types, the value of GALFUP remains zero.

GALFUP is used in several places. For example, MAINS uses GALFUP to determine if the current problem program main parameter card has been followed by a $ control card. Procedures of the "SUB routine" class which do not have mnemonics beginning their subparameter statements (e.g. BALSUB, LPSUB, THRSUB, and TWOSUB) use GALFUP to find the end of their subparameter statements.

GALFUP is most conspicuously used in SCAN. As each card image is shifted into CARD, GALFUP is copied into GALF which is used to indicate the type of card currently being scanned. This information is used by SCAN to properly process the current card image.

G.3. SHIFT

Subroutine SHIFT, called only from PROLOG and SCAN, is called whenever it is necessary to copy a card image from one 80 halfword area into another. SHIFT has three arguments:

the target area
the area containing the image to be copied
an integer counter to be incremented

SHIFT is called as each image is shifted from the look-ahead buffer into the actual scanning buffer. Whenever SHIFT is called for this reason, the counter argument passed is INPUTC equivalenced to IPROG(19).

The other times that SHIFT is called is going into or coming out of a repeat sequence. In these instances a temporary storage area, TPCARD, is either the target area or the area from which the image is coming. In these instances, the actual card counter INPUTC is not used, but rather a dummy variable is incremented and subsequently ignored.

G.4. CNTROL

CNTROL is called whenever any of the subroutines in the Syntax Interpreter prints out a line. CNTROL keeps track of how many lines per page have been printed, and if the number of lines exceeds MAXLINES, CNTROL goes to a new page and prints out a page heading.

CNTROL has one argument which is the number of lines printed. The page heading at the top of each page has the current date, placed in entry THEDATE by subroutine SEARCH. The page heading also has the page number of that page within the Syntax Interpreter printout.

G.5. SYNTAX, NONE, NO$OPT, ECOUNT

This subroutine is a catchall of various Syntax Interpreter error monitoring routines.

Entry SYNTAX is called from one of the "syntax-unit" routines whenever a syntax error is found. The "syntax-unit" routine is responsible for printing out the particular syntax-unit type (i.e."ERROR. ADDRESS SYNTAX.," or "ERROR. FLOATING-POINT CONSTANT SYNTAX.," etc.). SYNTAX calls CNTROL to count the one line message printed by the "syntax-unit" routine. SYNTAX then increases ABORT by 1; note that, correspondingly, no calling program should increment ABORT if SYNTAX is going to be called to signal the error. SYNTAX then prints out the position of the scan pointer at the time the error was signalled. CNTROL is then called to count the lines in the scan pointer message and then SYNTAX returns to its caller.

Entry NONE is called to indicate "MISSING RIGHT DELIMITER." This typically happens if an ENDP or ENDS card is found in the middle of parameter scanning by a "syntax-unit" routine. At best, a right delimiter must be missing for an ENDP or ENDS card to appear during a "syntax-unit" parameter scan. Again, ABORT is incremented, CNTROL is called an appropriate number of times and a proper message is printed.

Entry NO$OPT is called by routines which have called INTERP to scan parameter arguments, but no arguments have been found when they are definitely required. Again, ABORT is increased by one, an error message is printed and CNTROL is called.

Entry ECOUNT is called immediately prior to exit from execution of the FORTRAN portion of the Syntax Interpreter. ECOUNT merely prints out an error summary (i.e. the value of ABORT, if non-zero) and returns to its caller.

G.6. SDUMP, SNEXT, SERROR

This subroutine provides an error exit and a snapdump facility which is one level higher than the actual ERROR and NEXT exits of the subroutine library. This permits the printing out of the number of cards read by the Syntax Interpreter independent of the kind of exit, SNEXT or SERROR, taken. SERROR is called on SUPER ERROR conditions. SNEXT is called as part of normal termination by subroutine SEARCH.

SDUMP was placed in this subroutine as a matter of convenience. SDUMP is called by subroutine SCAN whenever a #SNAP= or #SNAP,N= card is encountered. SDUMP returns to its caller.

H. COMMON AREAS

The Syntax Interpreter has the following common areas:

common area	arrays in common	type
Blank common	IPAR(96)	I*4
	EXPARM(8)	L*4
	IPROG(32)	I*4
	CARD(80)	I*2
	CARDUP(80)	I*2
	JPAR(1402)	I*4
	FMT(148)	I*4
SHARE	XARRAY(512)	I*4
TRACOM	DFLAG	I*4
	IDOUFL	I*4
	LCOUNT	I*4
	LABEL	I*4
	LABREF(24)	I*4
	ID(1000)	I*2
	LABELS(1000)	R*8
	BRANCH(400,3)	I*2
EXTRA	IPOINT(2001)	I*2
VCOM	VRAY(24,9)	I*2
	VLOG(9)	L*1
REP	REPAR(24)	I*2
SSPACE	SEARSI	I*4
	APGM	I*4
	LUNIT(100)	L*1
	LDISK	L*1
	WARNFL	L*1

Blank common contains the arrays most generally required within the Syntax Interpreter. IPAR, JPAR, and FMT are used to store parameters as they are scanned from parameter cards. EXPARM and IPROG contain the most important variables used to control execution of the Syntax Interpreter. CARD and CARDUP are used to store card images as they are read for scanning purposes.

The common area SHARE is used by the various SUB routines for building records of subparameter information to be written onto unit 3. TRACOM is used by TRASUB, INTERP, and RECODE for constructing special information needed by TRASUB. DFLAG is used for dealing with flagged conditions as found by INDEX. The remaining variables and arrays in TRACOM are used in handling label reference processing. The common area EXTRA appears only in TRASUB and was placed in a common area in order to avoid an apparent mis-compilation which otherwise resulted.

VCOM and REP are filled in by PROLOG in response to #V and #REPEAT cards. Each one merely holds the contents of the parameters as scanned from those cards. VRAY information is then used by UNIT and SCAN to handle V-type addresses. REP is used by SCAN to determine the length of repeat sequences.

Common area SSPACE is used to estimate storage requirements for each problem program to be executed. SEARSI contains the size of the SEARCH load module. APGM contains the contents of APGM in the SVT (see sections III.B. and V.B.). LUNIT is a logical array of 100 locations, each location corresponding to a DSRN. Corresponding locations in LUNIT are set to true if that DSRN is used by a problem program. LDISK is set true by subroutine UNIT if a DISK address is used. SSPACE is used by QUEUE to determine region requirements for dynamically allocatable programs. WARNFL is set true by MAINS at the beginning of execution of the Syntax Interpreter. If NOEXECUTE has been specified on the EXEC card, WARNFL is set false the first time a problem program is larger than the current region size. A warning message is printed by QUEUE that one

time. During the current job step, subsequent region faults will check WARNFL, which will now be false, and the warning message will be skipped. This prevents the same warning message from being printed more than once in a job step.

VI. Miscellaneous

A. EXIT

The special library program EXIT is invoked by the loader, UISOUPAC, if an error condition occurs while in the loader (see section II.A). The purpose of EXIT is to close the FORTRAN controlled data sets and then return to UISOUPAC via the monitor. For the most part, EXIT is abstracted from the MAIN macro. EXIT performs the following functions:

1. Calls the initialization entry in IBCOM#.
2. Sets up addressability between the I/O routines in the EXIT program and the I/O monitor.
3. Calls the program exit entry in IBCOM#. Calling this entry causes the FIOCS# and DIOCS# controlled data sets to be closed and causes control to return to UISOUPAC via the supervisor.

Note that ERROR must be provided as an entry point in EXIT since INTRFACE references ERROR; however, subroutine NEXT, containing the usual ERROR entry, is not part of the EXIT load module.

B. LKED Macro

Due to size limitations in the SYS1.SYSJOBQE data set, it became impossible to create the subroutine library in one job without resorting to the following subterfuge. All object modules are assembled or compiled into a single data set which is then link-edited. A second link-edit step is then used to sort out the various subroutines into separate partitions in the subroutine library. The mechanism by which the various subroutines are sorted out is by use of the linkage editor control cards

```
REPLACE
INCLUDE
ALIAS
NAME
```

The LKED macro is designed to create a file of control cards appropriate for input to the linkage editor. The actual card images are provided by the PUNCH pseudo-op in the assembler. As the LKED macro is assembled, the card images are generated in the output file of the assembler. Note that only cards generated by PUNCH statements go into the assembly output file since the only thing that is assembled is the one macro instruction. No actual assembly language instructions are in the assembly.

Assembly of the LKED macro consists of cycling through the macro argument list an entry at a time. For a given entry, REPLACE statements are generated for all other names, and finally INCLUDE, ALIAS, and NAME statements are punched for the current argument. This process terminates when the list of macro arguments has been exhausted. The macro is simply invoked by the mnemonic LKED with the macro arguments following separated by commas.

For example the statement

```
LKED A,(B,C)
```

would result in generation of the following statements:

```
REPLACE X
REPLACE B
INCLUDE INPUT(MAIN)
NAME    A
REPLACE X
REPLACE A
INCLUDE INPUT(MAIN)
ALIAS   C
NAME    B
```

The REPLACE X statement above is actually a set of several REPLACE statements which are placed at the beginning of each member specification to delete entry points from the IBM FORTRAN subroutine library which would otherwise appear as unresolved external references (e.g., routines such as DSQRT, DLOG, DEXP, DMAX1). These references are resolved at the time that the individual problem program is created in load module form. If DSQRT were left as an unresolved external reference (by not using a separate REPLACE DSQRT card) in the NEXT partition, say, then all problem programs when finally link-edited would have DSQRT included in the load module even though it might never be called by any of the subroutines in that module.

Note that as new subroutines are added to the library, a complete set of REPLACE cards of all the other entry points already in the library must be generated, as must an INCLUDE card, ALIAS cards if required, and a NAME card for the new member. Furthermore, a REPLACE card for the new member must be generated for each of the previous members already in the library. This process would be quite tedious without the LKED macro.

C. SORT Interface

An interface between SOUPAC and the IBM SORT/MERGE is provided by the catalogued procedure SOUPSORT. In general SOUPSORT is run between two SOUPAC jobsteps. The first SOUPAC jobstep is used to do any pre-processing of the input data and is also used to prepare the SORT/MERGE control cards. The latter function is performed by the UTILITY program in the SOUPAC library.

The actual SOUPSORT procedure differs from conventional SORT/MERGE in that two exit routines, namely E15 and E35, are required. The function of these two exit routines is to remove the header record before sorting, so that it doesn't get sorted into the middle of the file, and to replace the header record at the front of the file in the final output file. After the execution of SOUPSORT, the sorted file is ready for use as any other conventional SOUPAC temporary storage data set. A major drawback of the SOUPSORT procedure is that, since only one SORT control card can be prepared at a time by UTILITY, in general only one data set can be sorted with facility between any two given SOUPAC jobsteps. Note that when using SOUPSORT, all SOUPAC jobsteps after the first one should, in general, have a DISP=OLD parameter coded on the EXEC SOUPAC card as follows:

```
//  EXEC  SOUPAC,DISP=OLD
```

APPENDICES

APPENDIX A

I/O INTERFACE

The following is quoted from a paper written by John Ehrman in October 1966 for the Stanford Linear Accelerator Center (SLAC) Computation Group. The paper documents subroutine FIO999 and is titled "A Method for Implementing Core-to-Core Conversion" (GSG Programming Memo #13).

A particular I/O operation specified in a FORTRAN program results in an involved sequence of calls among several programs. In general, the sequence of operations is as follows:

1. The READ or WRITE (or other I/O statement) in the FORTRAN program is coded into a call to IBCOM which specifies the unit number, the coded format string to be used (if any), whether the operation is to be under format control or not, and so forth.
2. After recording pertinent information, IBCOM calls on FIOCS to initialize a number of items for the I/O operation to be performed. In particular, the unit number, the type (formatted or not), and the direction (input or output) of the operation are specified.
3. FIOCS determines whether the unit number is legal, and if so interrogates the "Unit Table" for that unit. (If none exists, FIOCS obtains information from IHCUATBL (constructed at SYSGEN time) and obtains space from the supervisor into which working information, the DCB, and DECB are placed; this is a Unit Table, and a pointer to it is placed in IHCUATBL.)
4. If the DCB has not been opened, FIOCS opens the file, obtains buffers of the proper length and number, and sets up pointers to them in the Unit Table.
5. If the operation is input, FIOCS reads the first record into a buffer. If the operation is output, FIOCS clears a buffer to zero (for non-formatted output) or blanks (for formatted output). The location and length of the buffer are then returned to IBCOM, which then prepares to convert items to or from the buffer.

6. IBCOM then returns control to the FORTRAN calling program. For each item in the I/O list (a single variable or an array name), a call is made to an appropriate entry point in IBCOM. Using the information established in steps 1 and 5 above, IBCOM performs conversions as required and starts filling or emptying the buffer provided by FIOCS.
7. If a buffer is exhausted before all items have been transmitted to or from it (for example, a "/" in a format, the end of the format is reached, or more items are to be transmitted in an unformatted operation than can be contained in one buffer-load), IBCOM calls on FIOCS to either write out the buffer (output) or to refill it (input). FIOCS then performs the desired action and returns to IBCOM the location and length of a newly-prepared buffer (which may be the same or different, depending on whether single or double buffering is specified).
8. IBCOM then continues conversions and transmissions between the buffer and the FORTRAN program under the control of the calls from the FORTRAN program; steps 6 and 7 are repeated as many times as necessary.
9. When the end of the I/O list in the FORTRAN program is reached, a call is made to an entry point of IBCOM which indicates that any partial buffers are to be ignored (on input) or transmitted (on output). If necessary, IBCOM calls on FIOCS to perform any additional I/O and then returns control to the FORTRAN program to continue processing.

For complete details on the actual calling sequences for IBCOM# and FIOCS#, refer to the IBM microfiche documentation on these programs. A thorough understanding of these calling sequences is essential to an understanding of the machinations of subroutine INTRFACE.

APPENDIX B

SUBROUTINE ARGUMENTS

The following is a brief summary of the most commonly used subroutines currently in the SOUPAC subroutine library. Arguments to subroutines are listed with a type and a description. Under type, arguments are classified by mode and length (e.g., I*4, R*8, L*4), and whether the argument is an input parameter, i, whose value is set by the calling program; an output parameter, o, whose value is set by the subroutine; or a parameter which is both input and output, io. The subroutines described here are broken down into four groups:

GROUP I: I/O ROUTINES
GROUP II: MATRIX INVERSION ROUTINES
GROUP III: EIGENVALUE AND EIGENVECTOR ROUTINES
GROUP IV: MISCELLANEOUS SUPPORT ROUTINES

GROUP I: I/O ROUTINES

DMIN Purpose: to read an entire matrix into a double precision matrix which is located in main memory and disk, and is referenced through the subroutine MANAGE, with its various entry points.
Example:

```
CALL DMIN (IA,NROW,NCOL,X,A,ROW)
```

Argument	Type	Description
IA	I*4 i	input address
NROW	I*4 o	number of rows of the data matrix
NCOL	I*4 o	number of columns of the data matrix
X	I*4 i	matrix "id" number
A	R*8 o	main storage area for data matrix
ROW	R*8	work area

DMOUT Purpose: to write an entire matrix from a double precision matrix which is referenced through MANAGE and its various entry points.
Example:

```
CALL DMOUT (IA,NROW,NCOL,X,A,ROW,LOWER,LMODE)
```

Argument	Type	Description
IA	I*4 i	output address
NROW	I*4 i	number of rows of the data matrix
NCOL	I*4 i	number of columns of the data matrix
X	I*4 i	matrix "id" number
A	R*8 i	main storage area for data matrix
ROW	R*8	work area
LOWER	L*4 i	lower triangular matrix indicator
LMODE	L*4 i	data item length indicator

DREAD Purpose: to read an entire data matrix from temporary storage into a double precision, memory contained, array.
Example:

```
CALL DREAD (IA,NROW,NCOL,MAX,A,ROW)
```

Argument	Type	Description
IA	I*4 i	input address
NROW	I*4 o	number of rows of the data matrix
NCOL	I*4 o	number of columns of the data matrix
MAX	I*4 i	row dimension of A
A	R*8 o	main storage area for data matrix
ROW	R*8	work area

DWRITE Purpose: to write an entire data matrix from a double precision array onto temporary storage.
Example:

```
CALL DWRITE (IA,NROW,NCOL,MAX,A,ROW,LOWER,LMODE)
```

Argument	Type	Description
IA	I*4 i	output address
NROW	I*4 i	number of rows of the data matrix
NCOL	I*4 i	number of columns of the data matrix
MAX	I*4 i	row dimension of A
A	R*8 i	main storage area containing matrix to be written
ROW	R*8	work area
LOWER	L*4 i	lower triangular matrix indicator
LMODE	L*4 i	data item length indicator

FSET Purpose: to set flag for F format on printing output matrices by DMOUT or DWRITE.
Example:

```
CALL FSET (IA)
```

Argument	Type	Description
IA	I*4 i	output address

HEADER Purpose: to read the header record for a data set without reading the first row of data.
Example:

```
CALL HEADER (IA,NROW,NCOL,LMODE)
```

Argument	Type	Description
IA	I*4 i	input address
NROW	I*4 o	number of rows of the data matrix
NCOL	I*4 o	number of columns of the data matrix
LMODE	L*4 o	data item length indicator

ROWIN Purpose: to read one row of data from cards or secondary storage into a single precision array.
Example:

```
CALL ROWIN (IA,NROW,NCOL,I,ROW,LAST,LMODE)
```

Argument	Type	Description
IA	I*4 i	input address
NROW	I*4 o	number of rows of the data matrix
NCOL	I*4 o	number of columns of the data matrix
I	I*4 i	index of the current row being read
ROW	R*4 o	area of storage into which data is to be placed
LAST	I*4 o	end of file indicator
LMODE	L*4 o	data item length indicator

ROWOUT Purpose: to write one row of data from a single precision array onto secondary storage. Output is by default in single precision.
Example:

```
CALL ROWOUT (IA,NROW,NCOL,I,ROW,LAST)
```

Argument	Type	Description
IA	I*4 i	output address
NROW	I*4 i	number of rows of the data matrix
NCOL	I*4 i	number of columns of the data matrix
I	I*4 i	index of the current row being written
ROW	R*4 i	area of storage from which data is to be written
LAST	I*4 i	end of file indicator

TWOIN Purpose: to read one row of data from cards or secondary storage into a double precision array.

Example:

```
CALL TWOIN (IA,NROW,NCOL,I,ROW,LAST,LMODE)
```

Argument	Type		Description
IA	I*4	i	input address
NROW	I*4	o	number of rows of the data matrix
NCOL	I*4	o	number of columns of the data matrix
I	I*4	i	index of the current row being read
ROW	R*8	o	area of storage into which data is to be placed
LAST	I*4	o	end of file indicator
LMODE	L*4	o	data item length indicator

TWOOUT Purpose: to write one row of data from a double precision array onto secondary storage.

Example:

```
CALL TWOOUT (IA,NROW,NCOL,I,ROW,LAST,LMODE)
```

Argument	Type		Description
IA	I*4	i	output address
NROW	I*4	i	number of rows of the data matrix
NCOL	I*4	i	number of columns of the data matrix
I	I*4	i	index of the current row being written
ROW	R*8	i	area of storage from which data is to be written
LAST	I*4	i	end of file indicator
LMODE	L*4	i	data item length indicator

GROUP II: MATRIX INVERSION ROUTINES

INVERT Purpose: to invert a double precision matrix which is referenced through MANAGE and its various entry points.

Example:

```
CALL INVERT (A,X,NROW,NCOL,DET,DETEXP,POSDEF,IOPT,EPSLON,&100)
```

Argument	Type	Description
A	R*8 io	main storage area for input and result matrices
X	I*4 i	matrix "id" number
NROW	I*4 i	number of rows of the data matrix
NCOL	I*4 i	number of columns of the data matrix
DET	R*8 io	fractional part of determinant, if zero on input, calculation of determinant is skipped
DETEXP	I*4 o	exponent part of determinant
POSDEF	I*4 i	flag to indicate check for positive definiteness
IOPT	I*4 i	flag to indicate search for pivot element
EPSLON	R*8 i	criterion for singularity
&100		label for branching on singularity

INVMAT and INVRSE Purpose: to invert a double precision matrix which is entirely memory contained. INVMAT and INVRSE are entries into the same subroutine. The only difference between the two is that INVMAT has one extra parameter, POSDEF.

Examples:

```
CALL INVRSE (A,MAX,NROW,NCOL,DET, DETEXP,IOPT,EPSLON,&100)
CALL INVMAT (A,MAX,NROW,NCOL,DET,DETEXP,POSDEF,IOPT,EPSLON,&100)
```

Argument	Type	Description
A	R*8 io	main storage area for input and result matrices
MAX	I*4 i	row dimension of A
NROW	I*4 i	number of rows of the data matrix
NCOL	I*4 i	number of columns of the data matrix
DET	R*8 io	fractional part of determinant, if zero on input, calculation of determinant is skipped
DETEXP	I*4 o	exponent part of determinant
POSDEF	I*4 i	(INVMAT only) flag to indicate check for positive definiteness
IOPT	I*4 i	flag to indicate search for pivot element
EPSLON	R*8 i	criterion for singularity
&100		label for branching on singularity

GROUP III: EIGENVALUE AND EIGENVECTOR ROUTINES

TRIDI, EIGVAL, and EIGVEC constitute a set of subroutines which calculate eigenvalues and eigenvectors. TRIDI is called first, then EIGVAL, and finally EIGVEC is called once for each eigenvector to be calculated from its corresponding eigenvalue. The original input matrix is input into TRIDI in the array R. This matrix is assumed to be square symmetric. All remaining arrays are simple vectors.

TRIDI Purpose: to tri-diagonalize a double precision, square symmetric matrix.

Example:

```
CALL TRIDI (NCOL,MAX,R,A,B,W1,W2)
```

Argument	Type	Description
NCOL	I*4 i	number of columns (and rows) of the data matrix
MAX	I*4 i	row dimension of R. When matrix is stored linearly, that is without "holes," then MAX = NCOL
R	R*8 io	main storage area containing matrix to be tri-diagonalized. Note R is modified by TRIDI.
A	R*8 o	returns new diagonal
B	R*8 o	returns new first off diagonal
W1	R*8	work area
W2	R*8	work area

EIGVAL Purpose: to calculate the eigenvalues of a tri-diagonalized matrix as created by TRIDI.

Example:

```
CALL EIGVAL (NCOL,MAX,M,E,A,B,W1,W2,IPS)
```

Argument	Type	Description
NCOL	I*4 i	number of columns (and rows) of the original matrix
MAX		not used for anything
M	I*4 i	flag to indicate order of sorted eigenvalues. M > 0 implies descending order M < 0 implies ascending order
E	R*8 o	eigenvalues
A	R*8 i	resulting A vector from TRIDI
B	R*8 i	resulting B vector from TRIDI
W1	R*8	work area
W2	R*8	work area
IPS	I*4 i	flag to indicate type of sorting of eigenvalues IPS < 0 implies sort on signed values IPS = 0 implies sort on absolute values IPS > 0 implies sorting skipped

EIGVEC Purpose: to calculate the eigenvector of a matrix given one of its eigenvalues.

Example:

```
CALL EIGVEC (NCOL,MAX,R,A,B,E,V,W1,W2)
```

Argument	Type	Description
NCOL	I*4 i	number of columns (and rows) of the original matrix
MAX	I*4 i	row dimension of R
R	R*8 i	R matrix as modified by TRIDI
A	R*8 i	resulting A vector from TRIDI
B	R*8 i	resulting B vector from TRIDI
E	R*8 i	eigenvalues from EIGVAL
V	R*8 o	calculated eigenvector
W1	R*8	work area
W2	R*8	work area

GROUP IV: MISCELLANEOUS SUPPORT ROUTINES

BCF Purpose: to test double precision floating point numbers with any one of six relational operators.

Example:

```
CALL BCF (OPR,FLOATA,FLOATB,&100)
```

Argument	Type	Description
OPR	I*4 i	coded relational operator
FLOATA	R*8 i	first comparand
FLOATB	R*8 i	second comparand
&100		label for branching if test succeeds

ERROR Purpose: error exit from a problem program. Note, a call to ERROR should always be preceded by a WRITE statement which prints out an appropriate error message, roughly understandable to the SOUPAC staff at least, hopefully understandable to most SOUPAC users.

Example:

```
CALL ERROR
```

MZERO Purpose: to provide a function subprogram to test a variable for -0. MZERO returns a value TRUE if the argument equals -0.

Examples:

```
LTEST = MZERO(X)
IF (MZERO(X)) GO TO 200
```

Argument	Type	Description
X	R norm.	floating point number to be tested. Argument may be either R*4 or R*8, so long as it is normalized.

TPARA Purpose: to read in problem program subparameters for those programs which have subparameters.

Example:

```
CALL TPARA (PARM,ISIZE,&100)
```

Argument	Type	Description
PARM	I*4 o	area into which parameters are to be read
ISIZE	I*4 i	length in single words of PARM
&100		label for branching on overflow of PARM

Several other subroutines are in the SOUPAC library as support routines and hence these names are also reserved:

ALLOC8
ASDF
BCNVT
CHKERR
COPY
DABTBL
DADSET
FREAD1
FREAD2
IOREW
MAIN
NEXT
OUTSET
PGMEND
RCOPY
READ
REWCOM
SEQCHK
TREAD
TYME
XADDR

APPENDIX C

SOUPAC STEP PARAMETERS

The following is a list of options currently permitted to a SOUPAC job step. Options are specified in the PARM field of the EXEC statement which invokes the job step. An option, or its alternative NO form may not be specified more than once in a PARM field. If no options are specified in the PARM field, the underlined options are assumed as default values.

1. EXECUTE or NOEXECUTE

   NOEXECUTE implies that the SOUPAC parameter deck, for which the Syntax Interpreter is to scan and build intermediate parameters, should not be executed. NOEXECUTE indicates that only a syntax check is to be performed. If EXECUTE is specified and no errors are found by the Syntax Interpreter, the job step will proceed. If EXECUTE is specified and errors are found by the Syntax Interpreter, execution of the step may continue depending upon whether LET or NOLET is also specified.

2. NOLET or LET

   If an error is found by the Syntax Interpreter and EXECUTE has been specified, execution will proceed only if LET was also specified. In this case, execution will proceed only through the last program processed which was completely error free. If NOLET was specified and errors are found by the Syntax Interpreter, execution will not be permitted.

3. LIST or NOLIST

   LIST indicates that all program cards are to be listed. NOLIST indicates that only the prolog section of the SOUPAC parameter deck is to be listed.

4. PGM or NOPGM

   PGM indicates that a complete SOUPAC parameter deck and data decks follow. NOPGM indicates that only the prolog section and data deck follow, and that the intermediate parameters are being provided by the user by over-riding the catalogued procedure. This implies that the user has previously run a SOUPAC job and has saved the two necessary data sets so that he may run the same program again.

   If any error is found by the Syntax Interpreter in the prolog section, the job step will not continue.

If the job step which generated the intermediate parameter data sets found syntax errors, execution of the job step in which NOPGM is specified will continue (if EXECUTE is specified) through the last program processed which was completely error free regardless of whether LET or NOLET was specified in either job step.

5. GEN or NOGEN

This option is used to control the listing of repeat sequences created by the use of #REPEAT, #SREP and #EREP cards (see Appendix D). When GEN is specified all generated copies of repeated sequences are listed. When NOGEN is used, the cards between an #SREP and #EREP pair are listed only once, the repeated copies of each repeat sequence are not printed. In particular, in order to suppress the printing of those repeat sequence lines which begin with the character '+', use NOGEN.

Examples:

To do just a syntax check:

```
// EXEC SOUPAC,PARM='NOEXECUTE'
```

To execute up to the first program found to have syntax errors:

```
// EXEC SOUPAC,PARM='LET'
```

Note that the PARMS may be listed in any order. For example, the following two EXEC statements are equivalent.

```
// EXEC SOUP,PARM='LIST,LET'

// EXEC SOUP,PARM='LET,LIST'
```

APPENDIX D

PROLOG CARDS

Described below are several # control cards which may appear in the prolog of a SOUPAC job. If any of these prolog cards are used, they must appear immediately after the SYSIN card. Within the prolog, however, these control cards may appear in any order. The Syntax Interpreter determines the end of the prolog when it reads a card which is not one of these types. All prolog cards have parameters and, therefore, must be terminated by a period. Prolog cards may not have continuation cards, hence all parameter information must be punched within 80 columns. There is no limit to the number of prolog cards permitted, nor is there any restriction on the number of any one type. If conflicting information is entered, the information entered last overrides any previous definitions.

#DEFINE

Whenever the user wishes to specify the dimensions of a direct access data set (DISK address), punch #DEFINE in the first seven columns of a card. Next punch the address, the number of rows, and the number of columns as parameters in the usual SOUPAC fashion. Include this card in the prolog section of your program. For double precision matrices, code the same number of rows, but twice as many columns as otherwise. If the user desires to use any DISK address, #DEFINE must be used (in addition to supplying the necessary DD cards).

For example, to define a data set for DISK 17 with 20 rows and 40 columns double precision, prepare the following card:

```
#DEFINE (DISK17) (20) (80).
```

#OLD

The #OLD option is used to define the number of rows in a sequential data set created by a previously run SOUPAC job step. The number of rows is entered into a table in the monitor. This option should be used whenever the header record on a data set is not known to have a correct value for the number of rows, and the user does not want to execute a MATRIX MOVE to count the rows.

To use this option, punch a card with #OLD in the first four columns. Then code the sequential address and the number of rows in the usual SOUPAC fashion. The number of columns may be coded on the card, if desired, but will be totally ignored by SOUPAC. Include this card in the prolog section of the SOUPAC program which will be reading the previously created data set.

For example, to indicate that a previously created data set to be input from S1 has 77 rows, prepare the following card:

```
#OLD (S1) (77).
```

#REPEAT

The #REPEAT option is used to repeat sections of a SOUPAC parameter deck an optional number of times. The #REPEAT card which appears in the prolog section may be followed by integer parameters, up to a maximum of twenty-two, each parameter indicating the number of repetitions for a given repeat sequence. The card sequences to be repeated will be preceded by a #SREP card and will be followed by a #EREP card.

Example:

```
//  EXEC  SOUPAC
//SYSIN DD *
#REPEAT (2).

     < additional program cards >

#SREP
CORRELATION (C) ( ) (S1).
SQUARE ROOT (S1) (P(F)) (20) (C) (P(F)).
#EREP

     < additional program cards >

ENDS
```

In this example, the program sequence of CORRELATION and SQUARE ROOT will be repeated twice. Notice that for each parameter on the #REPEAT card there is a unique corresponding #SREP , #EREP pair. The statements enclosed by the #SREP , #EREP pair are repeated the number of times indicated on the #REPEAT card. In the example above, four card input data sets would be required for the repeated sections. Why? One card deck will be required by CORRELATIONS each time it is executed, and similarly SQUARE ROOT will require a deck each time it is executed.

Repeat sequences which begin before a main program and end in a subprogram, or which begin and end in different subprograms are not allowed. Also, a #SREP card cannot be immediately followed by a #EREP card (i.e. no statements in the repeat sequence), and an unmatched occurrance of either a #SREP or a #EREP card is an error.

Example of improper use of #SREP , #EREP

```
MATRIX.
MOVE (C) (S1).
#SREP
ENDP
CORRELATION (S1) (P) (S2/P).
TRA (S2).
#EREP
DIV (2) (1) (101).
OUT (P) (101).
ENDP
```

#TEST

The #TEST option is used to run programs which have not been entered into the Syntax Interpreter. A membername TESTSLOT has been reserved in the Syntax Interpreter and is invoked by the mnemonic TEST. TESTSLOT has no parameters defined, but parameter types may be defined at run time by including a #TEST card in the prolog section of the SOUPAC program. To define parameter types, punch #TEST in the first five columns of a card. Next, code the integer arguments, as determined from the table given below, which indicate what parameter types TESTSLOT is to have for the current run.

Table of Parameter Types

integer coded on #TEST card	argument on TESTSLOT program card
1	address
2	fixed point constant
3	floating point constant
4	fixed or floating point constant
5	not available to TESTSLOT
6	relational operator
7	not available to TESTSLOT
8	index set

All types have a length of one full word except for type 8 which has a length of three full words. The implied total length of all parameters defined for TESTSLOT should not exceed twenty-two full words. Also, TESTSLOT allows for the use of $-control cards, but does not allow for the use of subparameter cards.

For example, to run a program called ABC, set up a deck as follows:

```
//  EXEC  SOUPTEST
//ASM.SYSIN DD *
         MAIN  ABC,01JAN72,'ABC TEST'
         END
/*
//FORT.SYSIN DD *
       SUBROUTINE ABC

          < your FORTRAN program >

/*
//LKED.SYSIN DD *
    ENTRY   MAIN
    NAME    TESTSLOT
/*
//SOUP.SYSIN DD *
#TEST (1) (1) (2) (2) (3) (1) (3).

          < additional SOUPAC statements, if desired >

TEST (CARDS) (S1/P) ( ) (1) *95.* (S2) *2.*.

          < additional SOUPAC statements, if desired >

ENDS

          < data decks, if required >

/*
```

#V

The #V option is used to specify an address type which can assume different address values. This option is most often used in conjunction with the #REPEAT option for addresses which are within a given #SREP , #EREP pair. Since SOUPAC does not allow actual program loops (except for loops within the TRANSFORMATIONS program), #V is used to provide the ability to "index" over an array of addresses as those addresses are generated by the Syntax Interpreter.

The form of #V can be represented in a generalized form as

$$\#Vn\ (m)\ (A_1)\ .\ .\ .\ (A_k).$$

where n is an integer from 1 through 9 (thus there can be at most 9 variable addresses, namely V1 through V9), and m is a counter. The counter m is used to indicate how many times the particular V address assumes a real address in its list before it goes on to assume the next real address as its value. A_1 through A_k are the real addresses which #V assumes in turn with the restriction that there can be at most twenty such addresses. Note, however, that if after the last address, A_k, has been used Vn occurs again in the program, Vn goes back to assume the value of A_1 again and continues as before.

Example of a program using #V and an equivalent program not using #V.

using #V

```
//  EXEC  SOUPAC
//SYSIN DD *
#V9 (2)(S1)(S2)(S3)(S4).
#V8 (1)(S6)(S7)(S8)(S9).
MATRIX.
#SREP
MOVE (C) (V9).
TRA (V9) (V8).
#EREP
VERT (V8)(V8)(V8)(V8)(S5).
ENDP
    <remainder of program >
```

not using #V

```
//  EXEC  SOUPAC
//SYSIN DD *
MATRIX.
MOVE (C) (S1).
TRA (S1) (S6).
MOVE (C) (S2).
TRA (S2) (S7).
MOVE (C) (S3).
TRA (S3) (S8).
MOVE (C) (S4).
TRA (S4) (S9).
VERT (S6)(S7)(S8)(S9).
ENDP
    <remainder of program >
```

APPENDIX E

ERROR CONDITIONS

The following is a list of the types of error conditions which can cause a SOUPAC job to fail.

1. ABEND U0012 Errors of this type are errors which have been found by SOUPAC written code. These errors are the direct result of a SOUPAC program executing a CALL ERROR statement. An example of an error fo this type is attempting to perform a MATRIX operation with two matrices which are not conformable under that operation. Another example is the case when sequence checking of an input deck is being performed, but the deck is out of sequence.

2. ABEND U0016 Errors of this type are errors which have been processed by the FORTRAN extended error monitor facility. Errors of this type will always have an IHCnnnI type error message also printed out, where nnn is a three digit decimal integer. The three digit number indicates the specific execution problem. For example,

   IHC212I IBCOM - FORMATTED I/O, END OF RECORD ON UNIT 5

   which means in SOUPAC terms that the user's data format card indicates a card longer than 80 characters, will result in a U0016. Examples of other types of errors which result in U0016 abends are underflow, overflow, divide check, format errors, data conversion errors, and improper arguments to IBM written subroutines, such as a negative argument to a square root function.

3. ABEND U0020 This class of errors is rather rare in frequency of occurrence. These errors arise whenever there is some immediate difficulty within the SOUPAC monitor, UISOUPAC. The usual reason for a U0020 is the inability of UISOUPAC to find a desired program in the program library.

4. System abends. Common examples are: 0C1 , 0C4 , 0C5 , 80A , 522 , B37 . The complete documentation of system abends, except for the locally written additions such as A22 , B22 , E22, can be found in IBM document C28-6631.

5. JCL errors. These errors will, in general, not allow the SOUPAC program to even begin execution. These errors usually result from misspelled keywords, blanks inserted or deleted incorrectly, or missing commas.

6. ID card errors. These errors will cause all remaining cards in the deck to "flush." These result from incorrect ps number, department, or codeword. ID card errors can also result from insufficient funds, extra blanks on the card where there should not be any, and missing commas where they are required.

APPENDIX F

MEMBER NAMES

The following is a list of the statistical procedures currently found in SOUPAC, with a list of corresponding program numbers.

1	2	TESTSLOT	MATRIX
3	4	CORREL	FREQUE
5	6	T$TEST	LINEAR
7	8	REGRES	PRINCI
9	10	VARIMA	COMMUN
11	12	OBLIMA	ITERAT
13	14	DISCRI	AUTOCO
15	16	TRANSF	BISERI
17	18	CANONI	CENTRO
19	20	CLASSI	CLIQUE
21	22	THREEM	FITNES
23	24	PAIRED	PARTIA
25	26	BALANO	MULTIP
27	28	STANDA	UTILIT
29	30	RESTRI	SQUARE
31	32	PROCRU	RANKOR
33	34	SCALOG	STEPWI
35	36	BINORM	MISSIN
37	38	TRNPRO	UNREST
39	40	ECONOM	THREES
41	42	PROBIT	QUADRA
43	44	RANDOM	JACOBI
45	46	K1CLAS	K2CLAS
47	48	KOLMOG	THEILG
49	50	PLOTOO	MANNWH
51	52	TWOSTA	CROSPA
53	54	MULPRO	RD1800
55	56	ORTPRO	VEWPTS
57	58	CCPLOT	LINPRO
59	60	KCLASS	COVFAC
61	62	PERSON	TETRAC
63	64	VARISI	POLYNO
65	66	PEGRAM	

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-72-529	2.	3. Recipient's Accession No.
4. Title and Subtitle SOUPAC SYSTEM PROGRAMMER'S GUIDE			5. Report Date June 1972 6.
7. Author(s) Paul Chouinard			8. Performing Organization Rept. No.
9. Performing Organization Name and Address Department of Computer Science University of Illinois at Urbana-Champaign Urbana, Illinois 61801			10. Project/Task/Work Unit No. 11. Contract/Grant No.
12. Sponsoring Organization Name and Address			13. Type of Report & Period Covered Research 14.

15. Supplementary Notes

16. Abstracts

SOUPAC is a system of statistical and data manipulative computer programs designed and written at the University of Illinois Urbana Campus. The purpose of this system is to provide the statistical researcher with a broad range of easily accessible programs capable of satisfying his particular informational requirement. The emphasis is on the system concept since it is here that the full power of SOUPAC is discovered. It is this system concept which is responsible for the relative ease of usage that SOUPAC has achieved vis-á-vis other statistical packages. This report is a documentation of the design of the SOUPAC system.

In particular it includes a description of the various system components, such as the monitor and I/O support routines, and how they interact. Also a description of the Syntax Interpreter is provided.

17. Key Words and Document Analysis. 17a. Descriptors

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report)	21. No. of Pages
Release unlimited	UNCLASSIFIED	168
	20. Security Class (This Page) UNCLASSIFIED	22. Price